I-MATH ASSOCIATES, INC.
Image Processing & Electro-Optical Analysis

230 Cattail Court
P.O. Box 560788
Orlando, Florida 32856
(407) 857-3213
Fax (407) 826-8915
74553.406@compuserve.com

Technical Office
95 E. Mitchell-Hammock Rd.
Suite 202
Oviedo, Florida 32765
(407) 977-0200
Fax (407) 977-9070

INFORMATION FUSION USING N-DIMENSIONAL HASHING

FINAL REPORT

Principal Investigator
Ronald Patton

8 September 1997

Sponsored by:

U.S. Army Research Office
Attn: Dr. Ming C. Lin, COTR
P.O. Box 12211
Research Triangle, North Carolina 27709-2211

Issued by U.S. Army Research Office

Contract No. DAAH04-96-C-0064

Effective Date of Contract: 3 September 1996
Contract Expiration Date: 2 August 1997
Reporting Period: 3 September 1996 – 2 August 1997

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	8 September 1997	Fianl 3 September 1996-2 August 1997

4. TITLE AND SUBTITLE
Information Fusion Using N-Dimensional Hashing

5. FUNDING NUMBERS
See Item 11

6. AUTHOR(S)
Ronald Patton and Harley Myler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
I-MATH Associates, Inc.
230 Cattail Court
PO Box 560788
Orlando, FL 32856-0788

8. PERFORMING ORGANIZATION REPORT NUMBER
97IR42

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Office
PO Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
ARO 35959.1-MA-ST1

11. SUPPLEMENTARY NOTES
STTR 96 T006 - Accounting Data 2162040-66A-7270
P665502.86100-2581 S31124 686100/6ST 35959

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approval for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

During Phase I, I-MATH Associates, Inc. and the NYU Courant Institute of Mathematical Sciences have developed algorithms and real-time software for fusion of 3D imagery and information. The fundamental technique is geometric hashing. Hashing is an efficient method for storing a very large set of models, representing various target types and poses, and then quickly determining which model best represents an unknown item, whose corresponding features are sifted through the hash table.

In its current form, hashing represents an object's (or scene's) feature values in a 2D table whose abscissa and ordinate correspond to the feature variables. Typically, such features are (x,y) geometric coordinates of key interest points about the object (scene). However, the features can be any basis function, including affine transforms of a rigid body, radius of curvature and tangent magnitude of curved objects, etc. Hence, hashing allows disparate types of information to be placed in a common table. The overall objective of this STTR is not just multidimensional pattern recognition, but rather maximum extraction of information from multiple sources, which may be dissimilar and perhaps not even imaging. Hashing directly supports such fusion, since multiple types of features can be the basis for an nD hash table.

14. SUBJECT TERMS
STTR Report
N-Dimensional Fusion
2D Iterative Projection
Geometric Hashing

15. NUMBER OF PAGES
106

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

J51171
04-08-96

The thrust of this STTR development has been to devise an nth order hashing schema, beginning with a 3D implementation for Phase I. However, our approach is not limited to extending the hash table from a 2D to 3D (or higher dimension) domain. We have also investigated alternative techniques during Phase I, including:

- Hashing on 2D plane orthonormal projections, and then combining the results using postclassifier fusion techniques
- Once the best 2D match is made, then rotating the matched plane to the other orthogonal planes for matching refinement by performing additional hashing
- Combinations of the above.

Initially, we sought to port the ND Hash software developed by NYU into the Texas Instruments Multimedia Video Processor TMS320C8X parallel DSP system (C80). This task was unsuccessful due to the large amount of intra-memory accesses performed by the NYU algorithm. However, we were able to implement a 2D hashing algorithm onto a simulation of the C80. This provides a basis for hosting the iterative 2D projection hashing algorithms.

PREFACE

This Phase I SBIR project was performed by I-MATH Associates, Inc., for which the Principal Investigator was Mr. Ronald Patton. Significant contributions made by other I-MATH personnel were the implementation of 2D hashing onto the TI C80 processor (Dr. Harley Myler, with assistance from a UCF graduate student, Mosleh Uddin), the feasibility analysis of the Iterative 2D Projection Hashing algorithms (Liviu Voicu), and processor configuration evaluations, particularly the GAPP/PAL technology (Herb Arkin). New York Univeristy developed the Generalized ND Hashing software and demonstrated its feasibility with several examples (graduate student Raju Jawelekar, under the direction of Dr. Robert Hummel).

The ARO COTR was Dr. Ming C. Lin, (919) 549-4256, lin@aro.ncren.net. In addition to this Final Report, interim results were presented at the ARO Principal Investigators Meetings on Computational Mathematics, Discrete Mathematics, and Computer Science on 26-27 February 1997; Dr. Myler and Mr. Jawelakar were the presenters. A Phase II proposal was submitted on 28 May 1997.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

During Phase I, I-MATH Associates, Inc. and the NYU Courant Institute of Mathematical Sciences have developed algorithms and real-time software for fusion of 3D imagery and information. The fundamental technique is geometric hashing, originally conceived by NYU[1,2]; I-MATH subsequently applied geometric hashing to critical military applications.[3,4] NYU has a close liaison with the TJ Watson IBM Research Center, a significant hashing contributor.[5]

In its current form, hashing represents an object's (or scene's) feature values in a 2D table whose abscissa and ordinate correspond to the feature variables. Typically, such features are (x,y) geometric coordinates of key interest points about the object (scene). Some examples for Ladar imagery are shown in Figure 1. However, the features can be any basis function, including affine transforms of a rigid body,[6] radius of curvature and tangent magnitude of curved objects,[7] etc. Hence, hashing allows disparate types of information to be placed in a common table.

Hashing is an efficient method for storing a very large set of models, representing various target types and poses, and then quickly determining which model best represents an unknown item, whose corresponding features are sifted through the hash table. NYU has evaluated a number of computer architectures, including a Connection Machine for large scale hashing implementations.[8]

The overall objective of this STTR is not just multidimensional pattern recognition, but rather maximum extraction of information from multiple sources, which may be dissimilar and perhaps not even imaging. Hashing directly supports such fusion, since multiple types of features can be the basis for an nD hash table. In fact, features do not have to have the same dimensionality, the formulation for which has been developed at NYU.[9]

---

[1] Y. Lamdan and H.J. Wolfson, "Geometric Hashing: A General and Efficient Model-Based Recognition Scheme" Proc. 2nd International Conference on Computer Vision (ICCV), pp. 238-249, December 1988.
[2] R. Hummel and H. Wolfson, "Affine Invariant Matching," DARPA Image Understanding (IU) Workshop, April 1988.
[3] A. Akerman III, R. Patton, et al, "Multisensor Target Acquisition/Target Recognition Using Genetic Algorithms and Geometric Hashing," Proc 4th Automatic Target Recognizer (ATR) Science and Technology Symposia, ATRWG PR-44-001, Vol. I, pp. 61-90, March 1995.
[4] A. Akerman III, R. Patton, W. Delashmit, and R. Hummel, "Target Identification Using Geometric Hashing and FLIR/LADAR Fusion," ARPA IU Workshop, February 1996.
[5] A. Califano, "Multidimensional Indexing for Recognizing Visual Shapes," IEEE Trans on Pattern Analysis and Machine Intelligence, 16(4), 1994, pp. 373-392.
[6] I.Rigoustos and R. Hummel, "Several Results on Affine Invariant Geometric Hashing" Proc 8th Israel. Conference on Artificial Intelligence (AI) and Computer Vision (CV), Tel Avid, December 1991.
[7] E. Kishon and H. Wolfson, "3D Curve Matching, Proc of AAAI Workshop on Spatial Reasoning and Multisensor Fusion, pp. 250-261, October 1987.
[8] O. Bourden and G. Medioni, "Object Recognition Using Geometric Hashing on the Connection Machine," Proc International Conference on Computer Vision (ICCV), 1990.
[9] D. Geiger, R. Hummel, et al, "The Feature Transform for ATR Image Decomposition," SPIE Aerospace Symposia, Orlando, April 1995.

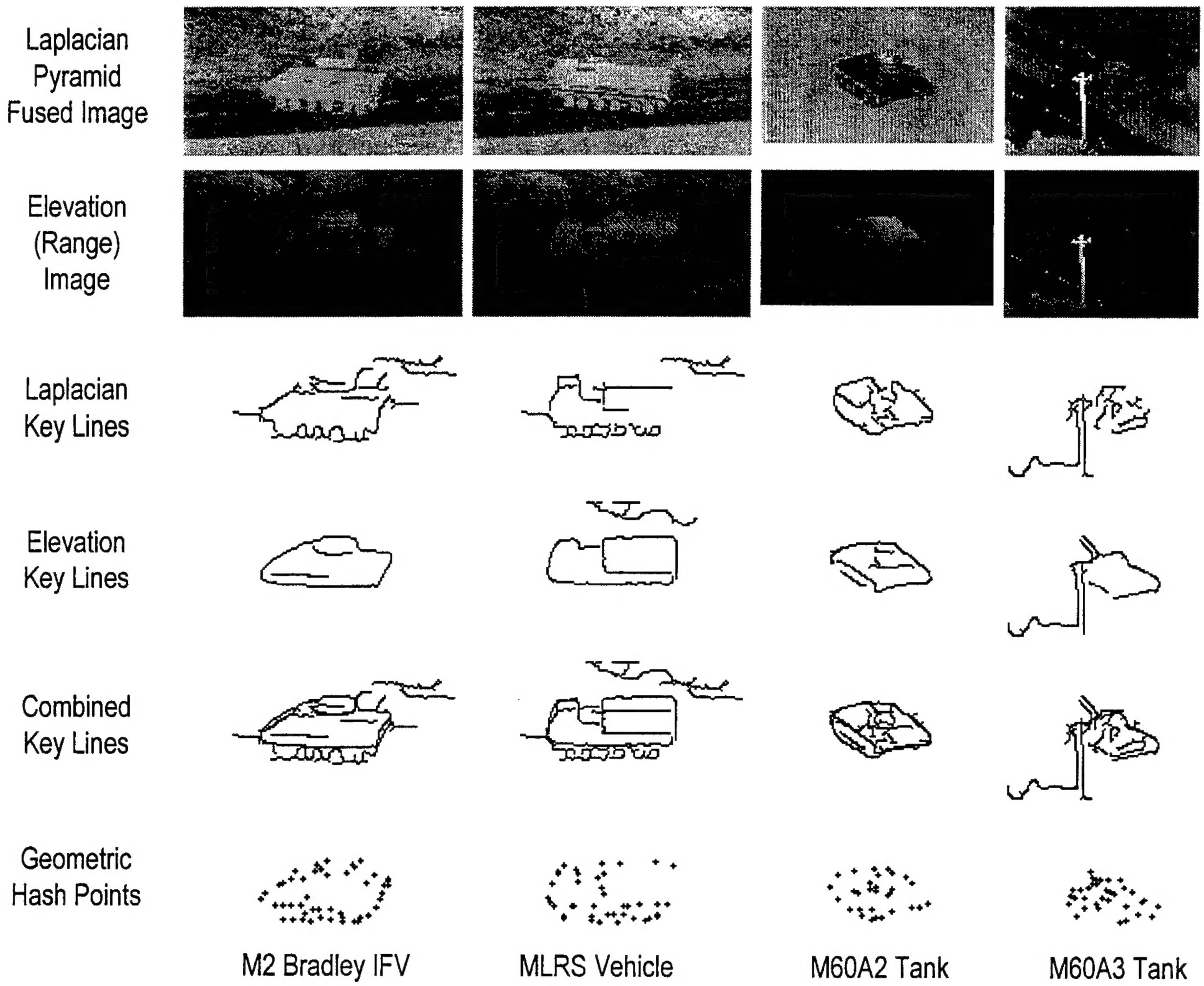


Figure 1. Geometric Hash Point Extraction Algorithms Applied to Ladar Imagery

The predominant nature of all the aforementioned programs and developments has been 2D hashing. At best, extensions to higher dimensionality has been achieved by appending labels to the 2D (x,y) hash points.[10] For example, our personnel were significantly involved in an ARPA/ARO program which includes Ladar ATR processing.[11] The software, which runs on a SUN SPARC 20 workstation, has been modified to attach Ladar range and intensity (and up to 8 other parameters) to the azimuth and elevation coordinates representing the key geometric components of the object. However, only the geometric components are hashed, with the unknown's labels then simply threshold tested against the corresponding models labels.

The thrust of this STTR development has been to devise an nth order hashing schema, beginning with a 3D implementation for Phase I; this is discussed in Section 2. However, our approach is not limited to extending the hash table from a 2D to 3D (or higher dimension) domain. We have also investigated alternative techniques during Phase I, including:

- Hashing on 2D plane orthonormal projections, and then combining the results using postclassifier fusion techniques
- Once the best 2D match is made, then rotating the matched plane to the other orthogonal planes for matching refinement by performing additional hashing
- Combinations of the above.

Those techniques are discussed in Section 3.

Initially, we sought to port the ND Hash software developed by NYU into the Texas Instruments Multimedia Video Processor (TMS320C8X) parallel DSP system. This task was unsuccessful due to the large amount of intra-memory accesses performed by the NYU algorithm. This algorithm makes extensive use of referenced data structures to represent the hash database and we were unable to port these constructs using the C80 simulator. In essence, the database is one large tree of pointer references where the coordinates of hash points and their relationships are pointers to other pointers. The database is traversed and search is accomplished by following the references down the tree formed by the structure pointers. This approach to the programming of the database was not consistent with the C80 architecture, which uses localized data spaces and was unable to process the large number of off-chip references into main memory.

However, we were able to implement a 2D hashing algorithm onto a simulation of the C80, as discussed in Section 4. This provides a basis for hosting the iterative 2D projection hashing algorithms described in Section 3.

There are a number of non-experimental parallel processing systems which potentially meet these ND Hash STTR criteria: 1) the parallel processing system is available in the commercial marketplace; 2) the system is intended to be general purpose in the sense that it is programmable for multiple applications; and 3) the system must be compact, meaning that it is

[10] Jyh-Jong Liu, A Model-Based 3D Object Recognition System Using Geometric Hashing with Attributed Features, NYU PhD Thesis, October 1995.

[11] A. Akerman, R. Patton, et al, "Geometric Hashing for Three Types of Sensor Imagery," 5th ATR Systems and Technology Symposium, John Hopkins University, July 1996.

implementable as a single-circuit card or as a collection of small cards in a module. The parallel processing systems that meet these criteria are listed in Table 1.

Table 1. Parallel Processor Candidates for ND Hashing

System	Company	Description
TMS320C80	Texas Instruments	MIMD Digital Signal Processor (DSP)
PAL-1/2	Lockheed Martin	Geometric Arithmetic Parallel Processor (GAPP)
MaxPCI	Datacube, Inc.	MISD (pipelined) system
HDS SR4300	Hitachi Data Systems	Scalable RISC processors in configurations ranging from 2 to 128 nodes.
MM32k	Current Technology, Inc.	SIMD architecture containing 32768 Processing Elements on a single AT board. Each PE is a complete fixed point 1 bit ALU with it's own 512 bit on-chip local memory.
A236	Oxford Micro	Parallel DSP
mPACT/3000	Chromatics Research	SIMD with very large crossbar switch in multiport RAM.

The processors listed in Table 1 that most readily supported some form of geometric hashing were determined to be the C80 and the Lockheed Martin Parallel Array Logic (PAL-1/2). As already mentioned, the C80 was proven feasible during Phase I for implementation of the iterative 2D projection form of hashing. Based on in-depth discussions with the Lockheed Martin PAL developers, it appears that the PAL-1/2 would be suitable for implementing the NYU Generalized ND Hashing algorithms. Hence, Section 5 provides additional detail on that PAL technology. A synergistic attribute of this approach is that NYU and Lockheed Martin have already worked together on another parallel processing application.[12] Both the C80 and the PAL processors should also be capable of hosting the attributed label form of 2D hashing.

The Datacube MaxPCI system is a parallel pipeline architecture and would not be as suitable for processing as either the C80 or GAPP because of the intra-pixel processing required. The Hitachi HDS SR4300 is a general purpose MIMD architecture of coarse scale that would be difficult to implement compactly. Additionally, software tools for parallel programming of the HDS SR4300 are unproven and not at the level of sophistication of either the C80 or the GAPP. Current Technologies MM32k system is a SIMD system very similar in architecture to the GAPP, and so offering no gain as an implementation vehicle. Likewise, the Oxford Micro A236 and the Chromatics Research mPACT/3000 systems are chip-systems like the C80. The mPACT/3000 is optimized for motion estimation and would not support hashing to the same level as that of the C80. The A236 is similar in the sense of the parallel DSP aspects of the C80, but does not have the same range of algorithm flexibility because it lacks a RISC-based executive processor.

[12] "The Fulcrum Project," NYU Final Status Report, July 31, 1996, http://cs.nyu.edu/phd_students/raju/fulcrum/fulcrum.html.

We have recommended that all three forms of ND Hashing be pursued in Phase II in the following manner:

1) 2D Hashing with Ten Attributed Labels - This is the form of the hashing software successfully demonstrated in the recent DARPA unmanned ground vehicle Demo II program for both FLIR and LADAR automatic target recognition.[13] This would be the baseline algorithms used for implementation on the PAL-1/2 processors.

2) Iterative 2D Hashing Projections - The feasibility of this algorithm was demonstrated during Phase I, both graphically and through implementation of the matching function on the C80 processor.

3) Generalized ND Hashing - NYU will continue the development of these Phase I algorithms with respect to implementation onto a PAL-1/2 processor. This will be accomplished using a simulation already at NYU from a previous collaborative effort with Lockheed Martin.[12]

It is important to note that the Phase II implementation encompasses all the geometric hashing processing functions shown by Figure 2, and not just the matching function. Accordingly, both the C80 and the PAL-1/2 will be evaluated with respect to each function. The resulting processor hardware may well be a hybrid combination of both the C80 and PAL-1/2. In addition to perform all the hashing functions, our goals for that hardware implementation are:

- Able to support a variety of sensor types
- Size less than 1.0 cubic foot
- Cost less than $20,000
- Capable of identifying >10 targets with a 90% correct probability
- Operation at 10-30 image frames/second

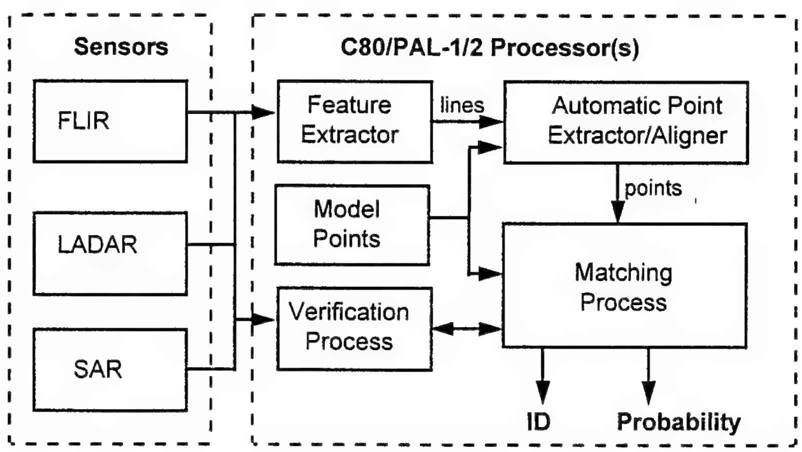


Figure 2. Architecture of the Geometric Hashing Processing Functions

[13] R. Patton et al, "Target Identification Using Geometric Hashing and FLIR/LADAR Fusion," in Chapter 4, "Target Detection and Recognition," Reconnaissance, Surveillance, and Target Acquisition for the Unmanned Ground Vehicle, Oscar Firschein, ed., Morgan Kaufman Publishers, Inc., in press.

2. GENERALIZED ND HASHING SOFTWARE

A major activity during Phase I was the rewriting of the hashing code by NYU to allow an arbitrary dimensionality. Initially, their weighted voting form of hashing was implemented in the ND structure. This is described further in Section 2.1. Various validation tests were performed using 3D Ladar imagery, which is further described in Appendix B. The validation test results were extensive, albeit limited to translation invariant cases. Section 2.2 summarizes those results. NYU subsequently enhanced the code to encompass rotation invariance about all three axes. A source code listing is given in Appendix A.

2.1 Summary of the ND Hashing Algorithm

When using geometric hashing for object recognition, there are two phases: The hash table construction (or preprocessing) phase, and the recognition (or on-line) phase. First, we discuss the construction of the hash table that will enable efficient object recognition using Ladar data.

The hash table encodes information about the models. For the NYU experiments, there were four different vehicle types, each sampled at 12 azimuthal directions, and at one depression angle. In order to accomplish the translation invariance, we use the notion of a basis feature. In our case, the features are 3D relative locations of boundary points on the ladar image of the model, sampled in such a way that we retain only those points whose 2D projection of the boundary exhibits high curvature at the corresponding image point. The 3D coordinate is computed relative to one of the extracted 3D points, which is the basis point. (See Appendix B for a listing of those hash points).

To construct the hash table, we do the following: For every vehicle type, for every view direction, for every possible basis point, we compute the collection of extracted relative 3D feature locations; for each such point, we construct an "entry" and include pointers to the entry in bins in a 3D hash table. We place a pointer to the entry in every bin that is "near" the 3D location of the feature.

As shown by Figure 3, the entry contains the following: The 3D relative location, knowledge of which model (type and view direction) and which basis point gave rise to the entry, and information necessary to compute a metric that measures the distance from the feature location to potential matching test points. This metric is based on the predicted statistical behavior of the model feature point.

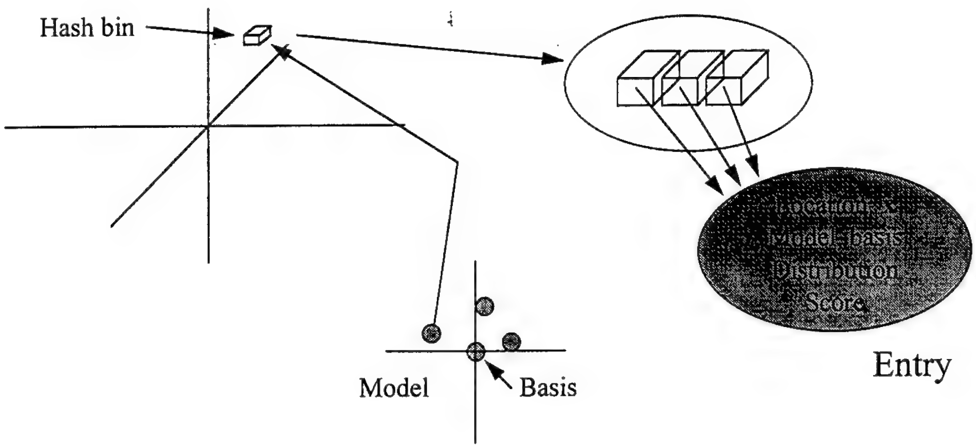


Figure 3. Hash Table Construction Methodology

In the recognition phase, we begin with an observed Ladar image, and we extract boundary points with high curvatures at the 2D projections. A basis point is chosen. The extracted features are computed relative to the basis point. For each such extracted feature, we have a relative 3D location, which "hashes" to a single bin in the hash table. In the hash bin, there is a list of pointers to entries. Each such entry is accessed and a distance between the observed relative 3D location and the entry's relative 3D location is computed. The value is stored in a "score" field associated with the entry by applying a max operator to the existing score. Initially, all such fields have a negative weight constituting a penalty for an unmatched model point. The process is shown in Figure 4.

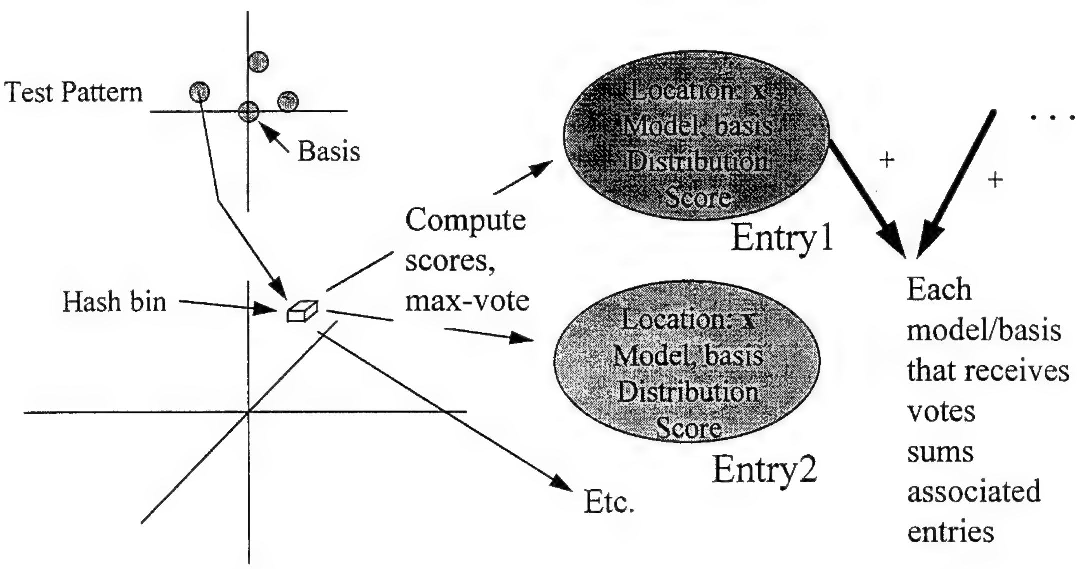


Figure 4. Voting Methodology for Determining Candidate Matches

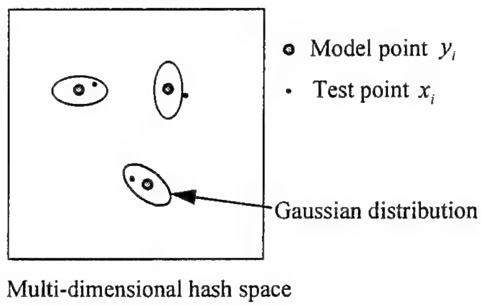
Whenever a score field is increased significantly, the associated model/basis which gave rise to the entry is "marked." After all max-votes are applied for all extracted features, then every marked model/basis computes a score by summing the scores of the entries associated with the model/basis. Note that there are many possible model/bases, since the model is a vehicle type at a particular view angle, and the basis is a designated feature extracted from the model. Finally, the top few model/basis scores are tallied and reported. These are candidate matches.

As detailed in Appendix C, NYU has evolved a theory to optimize and validate the scoring function that is applied to individual matches between features extracted from the test scene and features from the models. The formulas are based on a Bayesian computation of a log-probability of the match hypothesis relative to the prior probability, and also on an assumption of independence of the individual mismatches of the features conditioned on the match hypothesis.

Using Bayes' formula, the log-probability ratio can be converted into a log-likelihood ratio, which in turn can be seen to be a sum of log-likelihood ratios, of the from $\log(P(E \mid H)/P(E))$, where E is the "evidence" consisting of the proposed matches, and H is the hypothesis, i.e., the model. The numerator is the likelihood of a particular feature value given its association to a predicted model feature, while the denominator is the likelihood of the same extracted feature in the absence of a hypothesis of the presence of a model.

For the ND Hash experiments, we assumed that the predicted model features, computed relative to a basis point, will be distributed according to a Gaussian distribution in 3D space, with a characteristic covariance matrix that can be specified by the user. Accordingly, the log likelihood ratio of the individual summed terms that comprise the score for a matching hypothesis are each given by a log-ratio, or equivalently a difference of logs, using the Gaussian distribution for the matching hypothesis, and a background clutter density statistic for the a priori likelihood.

Finally, if the resulting score becomes too negative, then we can assume that the extracted feature should not be matched to the predicted feature, and we invoke a cut-off penalty for the unmatched model point. Likewise, if the score is too positive, then we use a clip value to ensure that no single match dominates the score. Figure 5 illustrates this Weighted Voting formula which is applied to the observed 3D point on the unknown test object and the 3D model points stored in the bins of the hash table.



Compute $\log\left(\frac{\Pr(\text{Model}|\text{Test\_data})}{\Pr(\text{Model})}\right)$

Assume independence of the errors

$\mu(x_i)$ clutter density distribution

$$Z = \sum_i \sigma_i \left[\log\left(G(x_i - y_i\right) - \log\left(\mu\left(x_i\right)\right)\right]$$

$\sigma_i(\cdot)$

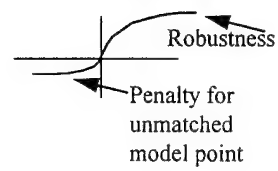


Figure 5. Weighted Voting Methodology

2.2 Validation Tests

As detailed in Appendix B, NYU began by constructing the four target models shown in Figure 6: the HMMWV configured as a cargo vehicle, the M113 APC, the M35 truck with a canvas cover over the truck bed, and the M60 tank. Using BRLCAD data, NYU generated synthetic Ladar data for each model at every angle azimuth sampled at 30 degrees. For testing purposes, a few models are generated at intermediate (15 degree angles) as well. NYU extracted points of high curvature. The location information together with the depth value became the 3D feature location.

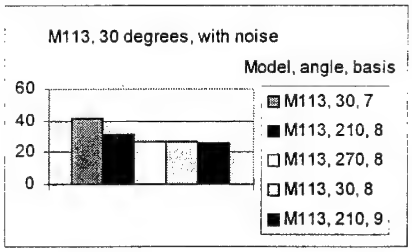
M35 canvas | M60 | HMMWV-Cargo | M113

Figure 6. BRLCAD Models Used to Validate the Generalized ND Hashing Software

A variety of experiments have been conducted to validate the software, including the following:

a. Using one of the models as a test target always produced a correct match. As an example, for the HMMWV (Cargo) model at 30° azimuth, the correct vote had the highest score of 74, whereas all other matches scored 30 or less.

b. Creating a new test target by adding uniform random noise to one of the models. With ± 0.5 units of random noise added to each hash point of the HMMWV case test model, the correct match was again made; not unexpectedly, the winning votes was somewhat lower at a vote of 60. As also shown by Figure 7, similar results were obtained for the M113 targets.



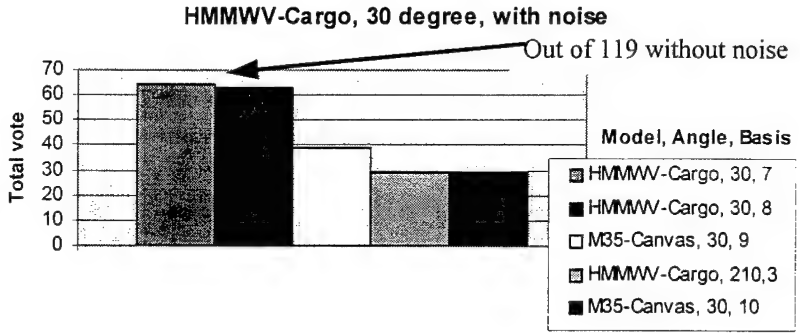


Figure 7. Validation Test Using Training Models with Noise Added

c. For each of the four target types in the hashing table an unknown target at 45° azimuth was tested to determine if it would be matched to the correct type, as well as the nearest model orientation (e.g., 30° or 60° azimuth).

(1) Unknown HMMWV (Cargo) - Matched to the correct model, with both the 30° and 60° orientations receiving the highest vote counts of 33.

(2) Unknown M113 - Incorrectly matched to HMMWV (Cargo)/270° azimuth and M60 tank/240° azimuth.

(3) Unknown M35 (Canvas) - Matched to the correct model for the three highest vote counts: 27-60°, 25-120°, and 24-30°.

(4) Unknown M60 - Incorrectly matched to a M35 (Canvas)/240° azimuth.

Although two of the tests results(paragraph c(2) and c(4)) are disappointing, they are not surprising for the following two reasons.

- I-MATH's experience with other types of sensor imagery has shown that good matching performance requires the model represent no more than 15° in angular excursion. In comparison, the NYU models are separated by 30°.

- I-MATH prefers that each target model have the same quantity of hash points, and that quantity be in the range of 40-50. As summarized in Table 2 below, the NYU hash table is much more sparse, particularly for the M113 and M60 which had the incorrect matches.

Table 2. Quantity of Associated Hash Points

	TEST	HASH TABLE	
	45°	30°	60°
HMMWV	27	20	23
M113	27	14	17
M35	32	24	28
M60	38	17	25

d. In a final set of experiments, two new unknown targets were created, both being variants of existing targets: the HMMWV was configured as a troop (rather than cargo) carrier, and the M35 truck had its canvas removed. These two variants are shown in Figure 8, which also shows the success of the ND Hashing code in matching the appropriate model.

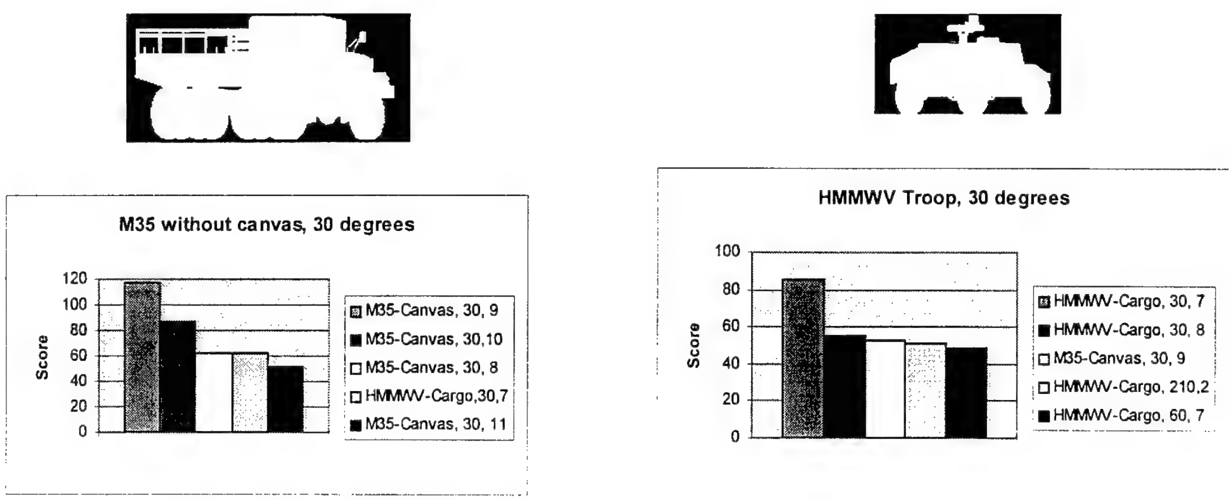


Figure 8. Validation Test Using Variants of HMMWV and M35 (HMMWV configured as a troop carrier, M35 without canvas)

3. 2D PROJECTION ALGORITHMS

3.1 Iterative 2D Projection Algorithms

We have developed an alternative to the existing ND Hash approach that we call iterative 2D Projection. This method makes use of the hash points derived from orthogonal projections of 3D targets rather than all of the 3D hash points used in the ND approach. There are computational efficiencies to be derived from this: consider the ratio between the number of comparisons performed in the ND case and those performed using 2D projections. This ratio is given by:

$$f(n,p) = \frac{C_n^2 C_p^2}{C_n^p}$$

where n is the dimension and p is the number of points per model. When the ratio is less than one, it takes more comparisons in the ND case and when the function is greater than one the 2D Projection method requires more comparisons. The function f(n,p) is a measure of computational demand and will be one for n=2 (the 2D Projection case) for any p. As n increases, the value of f(n,p) increases exponentially.

The 2D Projection approach represents a multiresolution method and because of this the computational efficiency can be exploited in a parallel architecture. The simplification of the hashing algorithm from a multi-dimensional requirement to multiple two dimensional problems will facilitate the implementation of the system, particularly in the C80. Our initial results indicate that the process is feasible.

The use of 2D Projections as a replacement for ND hashing opens up two important issues. The first is the observation that specific patterns may be distinguishable from each other only in one particular projection plane, as shown in Figure 9.

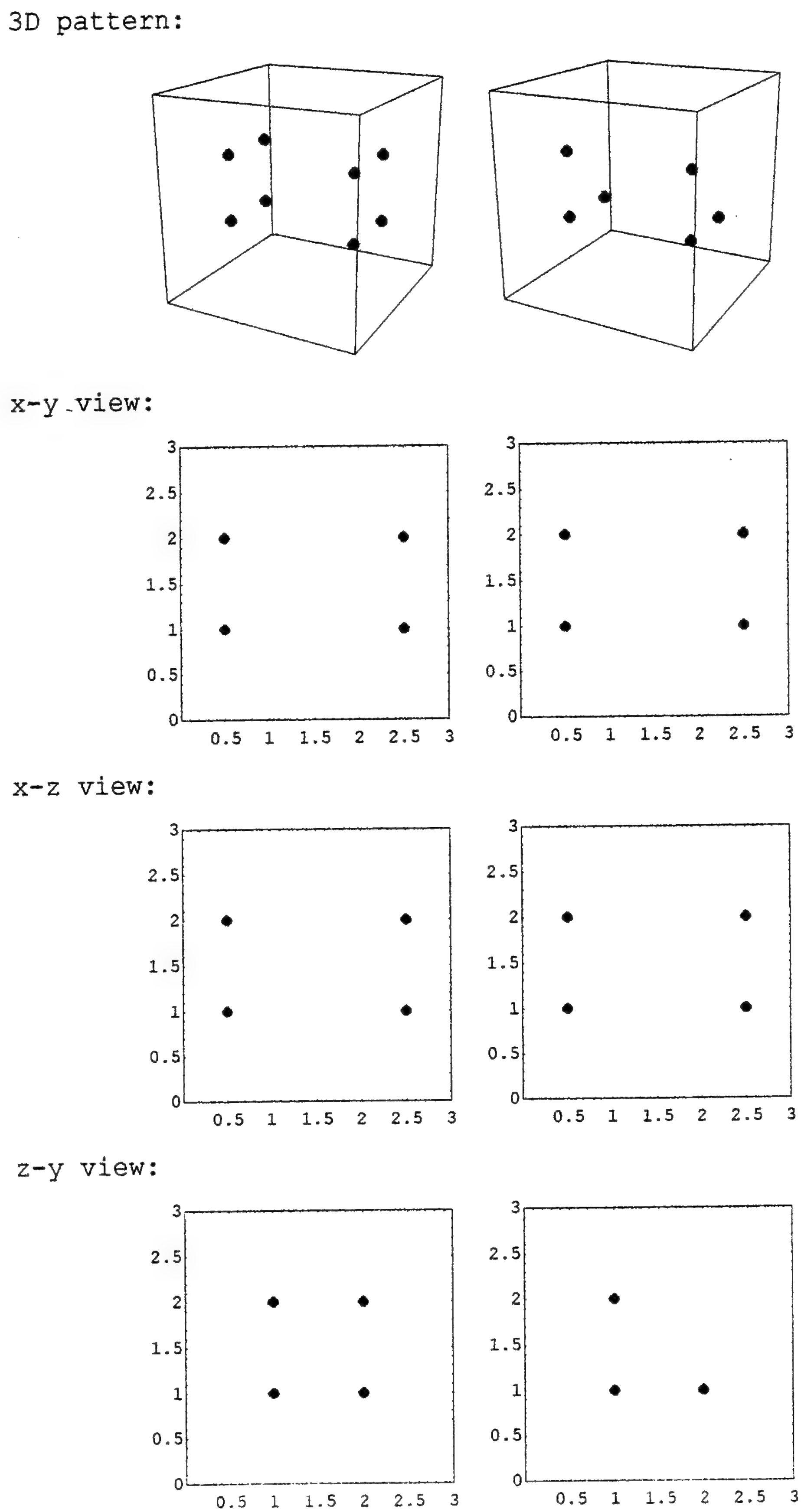


Figure 9. 2D Projections of Two Different 3D Objects Represented by Hash Points

Assume that the unknown is the prism object (2nd column in the figure) with each of its triangular faces represented by three hash points. Using a basic 2D Projection hashing scheme the rectangular model could develop the same voting weight that the prism model would yield, particularly if the points were slightly misaligned in the hashing bins due to noise in the source image. We propose a scheme that we call "2D Iterative Projection Hashing", or iterative projection for short. This involves labeling the points in both the unknown and model and monitoring subsequent pairings that occur at the different projections. The algorithm starts with a 2D hashing performed on the first projection. Then, the pairs of points that match are used as a filter for the next projection. Only points that matched previously are counted in the final vote. The point matches from the first projection that point match the second projection are passed to the third projection. A distance threshold will be employed that allows for a close-but-not-exact match The algorithm then outputs the number of pairs that matched across all projections. This approach will reduce errors due to false pairings and will also present a search advantage as only points that have previously matched are considered in the next iteration. Figure 10 summarizes the approach.

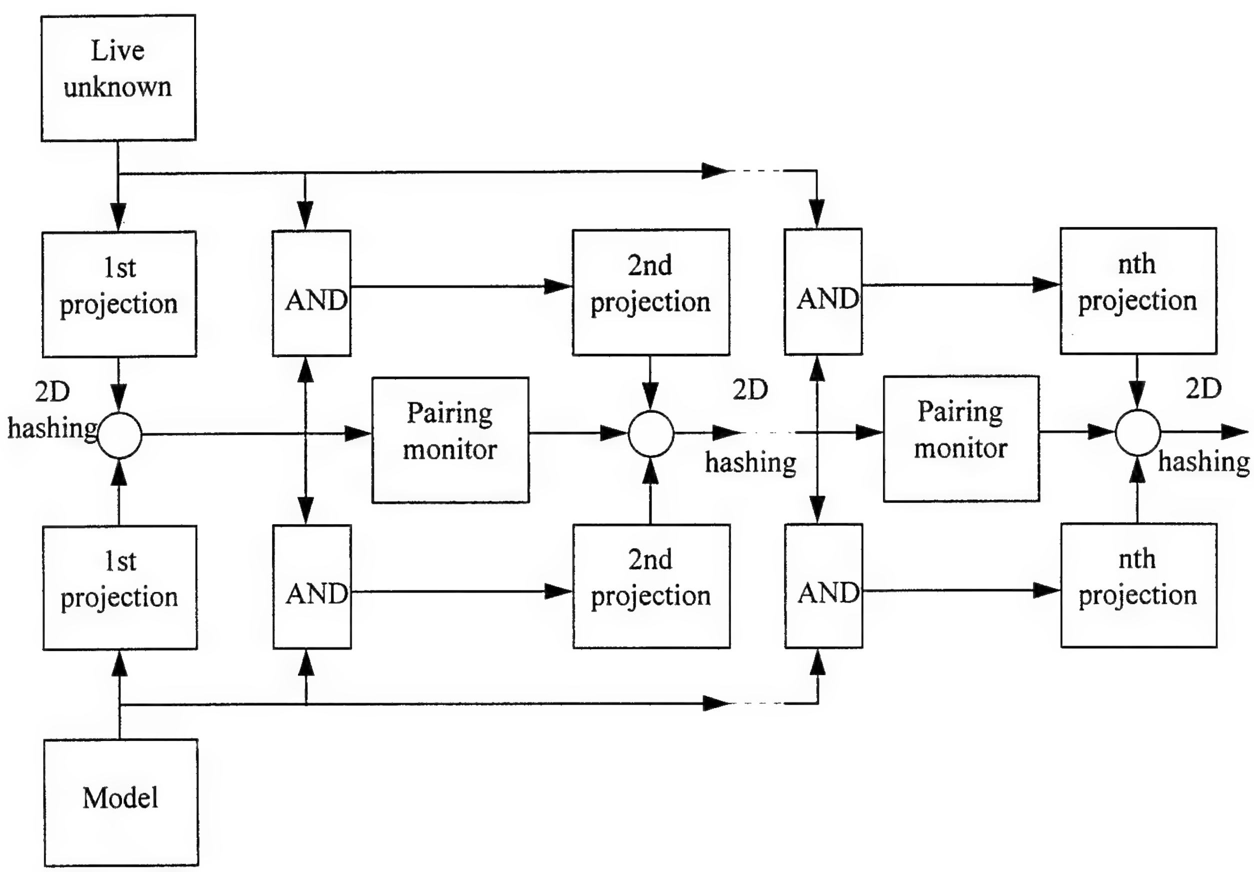


Figure 10. Iterative Hashing-By-Projection

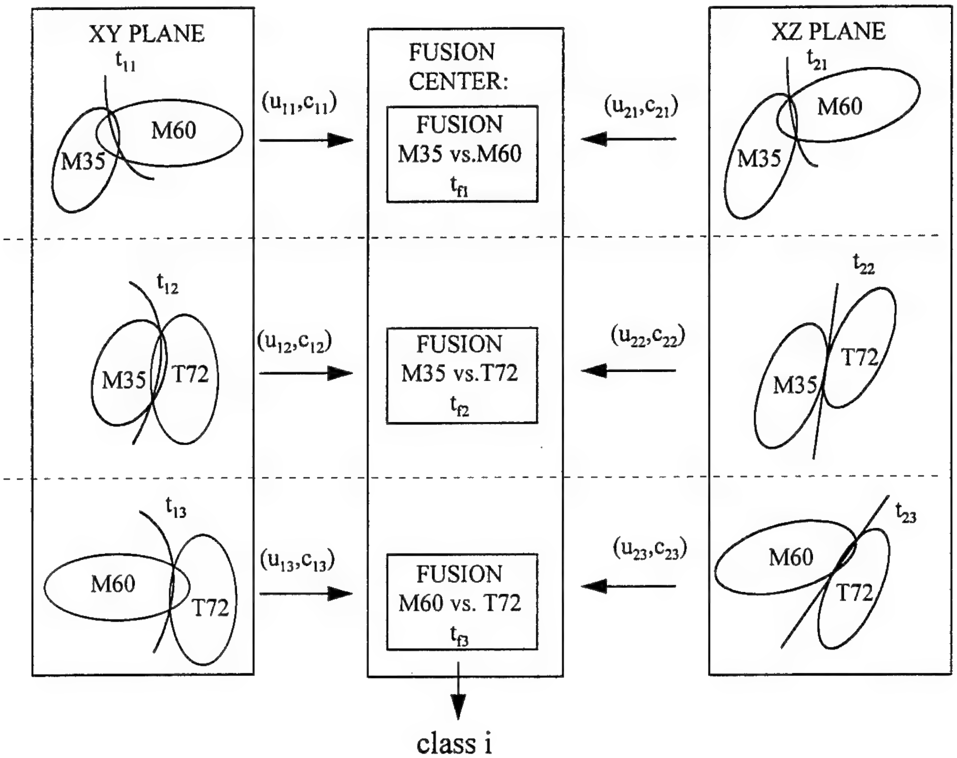


Figure 12. PLFA Decision Level Fusion Structure

The existing PLFA structure permits the conclusions of any number of hashing classifiers (each applied to an independent 2D projection) to be combined simply and reliably, using statistics that describe the classifier performance. These statistics must be provided n the form of ROC (receiver operating characteristic) curves which are graphs of the probability of correct versus incorrect decisions. The fundamental theory that describes the process by which these statistics can be fused was developed by Thomopoulos[16] and involves the application of a Neyman-Pearson (N-P) test using likelihood ratios established between the collaborating ROC curves. Thomopoulos showed how a N-P test could be applied at the sensor to generate a binary decision that could then be combined with other binary decisions at a fusion center.

[16] Thomopoulos, S., et al, "Optimal Decision Fusion in Multiple Sensor Systems," IEEE Trans. AES-023, No. 5, pp. 644-653, September 1987.

4. C-80 Implementation

4.1 Summary of the Technical Approach

Part of the Phase I effort was to port the ND Hash software developed by NYU into the Texas Instruments Multimedia Video Processor (C-80) parallel DSP system. This specific task was unsuccessful due to the large amount of intra-memory accesses performed by the NYU algorithm. The NYU algorithm makes extensive use of referenced data structures to represent the hash database and we were unable to port these constructs using the C-80 simulator. In essence, the database is one large tree of pointer references where the coordinates of hash points and their relationships are pointers to other pointers. The database is traversed and search is accomplished by following the references down the tree formed by the structure pointers. This approach to the programming of the database was not consistent with the C-80 architecture, which uses localized data spaces, and we were not able to program the processor to access the large number of off-chip references into main memory.

Our original strategy for the use of the C-80 was to take advantage of its four processor parallelism; however, these processors are optimized for the manipulation of pixel data that is cached in memory local to the processors. Any references to data in the main memory must be accessed through the master processor (An architectural discussion of the C-80 is given in Section 4.2). A bottleneck develops when there are extensive pointer references taking place between the four processors simultaneously. Additionally, the simulator was unable to link the NYU code because all memory references are handled as a cache fault and internal limits on this type of processing were reached. Nevertheless, we developed and were successful in implementing in the C-80 an alternative form of ND Hashing, 2D *iterative projections*; the details of that algorithm are given in Section 3.

The 2D iterative projection approach represents a multiresolution method and because of this the computational efficiency can be exploited in a parallel architecture. The simplification of the hashing algorithm from a multi-dimensional requirement to multiple two-dimensional problems will facilitate the implementation of the system in the C-80. Our results indicate that the process is not only feasible but desirable. Note that iterative 2D projections does not imply any loss of *n-dimensionality* as the space can be easily expanded across multiple 2D spaces.

An issue associated with 2D iterative projection is that of the multidimensional data beyond the 3D spatial projections that make the system truly "ND". The ND information can be treated as what we call "attributed labels". Attributed labels are data values that can be associated with a viewpoint or with a model point. In other words, if a LADAR intensity value is associated with a model point, then that intensity can be used as the fourth element of comparison. This feature is then hashed in the same way and provides an added vote to the final tally for an unknown. Additionally, if boresighted FLIR data is associated with a point, it can provide the fifth dimension of comparison. Since the comparisons are done independently (unlike the ND case, where all features are included in the same database), any *a priori*

influences to the fusion process that are desired can be applied external to the database at the algorithm level.

Fusion influencing thresholds are global to a feature type. For example, if there is intelligence that influences the detection probabilities of a sensor system, such as a weather forecast, the interpretation thresholds can be adjusted accordingly using the iterative projections approach. Alternately, if any match enhancing thresholds are stored as part of the database as in the ND case, then this will increase both the size and complexity of the database and force the search algorithm to use a single strategy for evaluation. Any changes to the information fusion process will have to occur in the database, or will necessitate a reconstruction of the database to match the new thresholds.

4.2 2D Parallel Hashing in the C-80

We were successful in implementing a 2D hashing algorithm in the C-80 that demonstrates our ability to produce the foundation of the 2D iterative projections algorithm. This was accomplished by coding the hash database for each model into a novel data structure that we call the Hash Database Image, or HDI. The HDI contains a compact representation of a hash table that is produced off-line. Each pixel coordinate in the HDI represents a key into the 2D hash table, while each pixel contains a coding of the model and basis pair associated with that coordinate. Each model translation and rotation are coded.

An HDI contains an entire projection database for a model and can be loaded and searched from a single Parallel Processor(PP). The advantage of processing the hash table into an image is that the C-80 is optimized for operations that involve pixel data and so the full power of the architecture is exploited. The search takes place in parallel over multiple classes, or multiple projections simultaneously. This strategy has a final bonus in that it is consistent with the iterative projections approach, which is computationally advantageous over the Generalized ND Hashing methodology. In the 2D hash case as implemented in the C-80, the Master Processor(MP) receives all votes produced by the Parallel Processors and tallies those votes to determine the target class of the unknown. (Each PP processes a single target class database). For the 2D iterative projection, each PP will process the first projection database of a different target class so that four classes can be examined simultaneously, as in the existing 2D hash case. After the initial projection, results will be passed to the next search, which will involve searching the filtered points (those points that matched) through the next projection. Points that pass this search will be searched in the final projection or passed through the attributed label datasets. Matches will then be sorted and formatted by the MP for final output.

As shown by Figure 13, the TMS320C80 Multimedia Video Processor (C-80) is a fully integrated parallel processor that is comprised of a 32-bit RISC Master Processor (MP) and four 32-bit Advanced Digital Signal Processors (ADSPs) on a single chip. The processor is driven by a 50 MHz clock that yields a 20 nsec basic instruction rate. Performance has been measured at 100 MFLOPS, 250 MIPS, and 2 BOPS overall. The 32-bit floating-point RISC MP is rated at 100 MFLOPS and 50 MIPS while the four 32-bit integer ADSPs are rated at 50 MIPS each.

The C-80 has 50 Kbytes of on-chip RAM, in 25 2-Kbyte blocks that are shared among all processors via a high-speed crossbar switch. The crossbar switch supports 15 concurrent accesses per cycle of 8, 16, 32, and 64-bit data. The C-80 is a fully programmable MIMD architecture. The system can access 4 Gbytes of off-chip RAM in the same data widths as the internal addressing. The MP has a 4-Kbyte RISC instruction cache and a 4-Kbyte data cache. Each PP has a 2-Kbyte instruction cache.

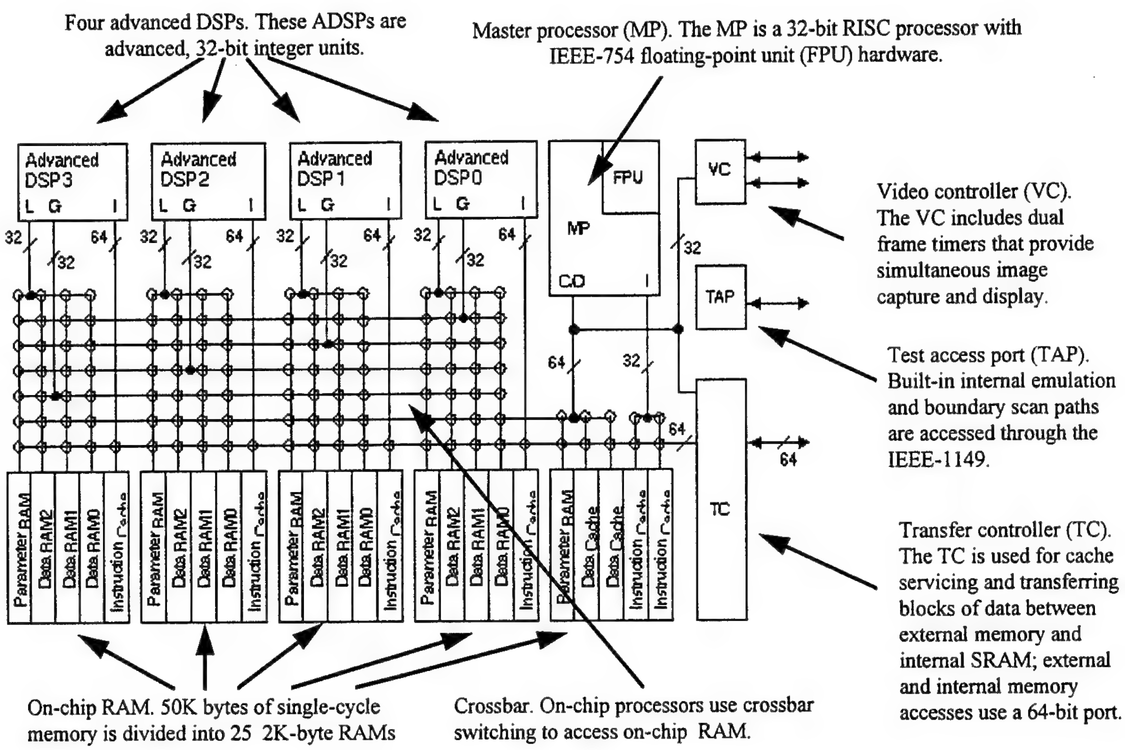


Figure 13. TMS320C80 (MVP) Architecture

The system can transfer 64-bit words at a rate of 400 Mbytes/sec via the on-chip transfer controller (TC). The system also has two video memory frame controllers on-chip to facilitate image I/O. The TC is a combined DMA machine and memory interface that intelligently queues, prioritizes, and services the data requests and cache misses of the MP and the PPs. The transfer controller interfaces directly with the on-chip SRAMs. Through the TC, all of the processors can access the system external to the chip. In addition, data-cache or instruction-cache misses are automatically handled by the TC. Data transfers are specifically requested by the PPs or the MP in the form of linked-list packet transfers, which are handled by the TC. These requests allow multidimensional blocks of information to be transferred between a source and destination, either of which can be on-chip or off-chip.

The parallel-processing advanced DSP (PP) data unit has two data paths, where each path has its own set of hardware that functions independently of the other. The ALU data path includes a barrel rotator, mask generator, 1-bit to n-bit expander, and a 3-input ALU that can combine the mask or expander output with register data to create over 2,000 different processing

options. The 3-input ALU can perform 512 logical and/or mixed logical and arithmetic operations that support masking or merging and addition/subtraction in a single pass.

All of these features make the C-80 an excellent vehicle for the processing of the 2D Iterative Projections algorithm. The C-80 processing of the Hash Data Images is not unlike discrete correlation and template matching, which the C-80 has been optimized to perform. The added capability of the Master Processor to the system allows for non-parallel processing such as vote tallies and application of nonlinear discriminants to take place independent of the parallel processing tasks.

We coded the 2D Iterative Projection hash algorithm in C code on a Sun computer that processes target points with selection of unique basis pairs and subsequent translation and rotation to the origin. The coordinate points are then stored as a table where the first column represents a coordinate in hash space and the second column the basis pairs associated with that coordinate. The table is then processed into a hash database image as discussed below.

4.3 Algorithm Implementation and Test Results

The system we designed to process 2D hashing in the C-80 is illustrated in Figure 14. The process begins with the formatting of a hash database into a database image. The hash databases, one per target class, are processed off-line using a known model set. Each entry in the database constitutes a coordinate pair with a set of matching pose points. These pose points are then encoded in binary and formed into a database 'image'. Each pixel in the image is coded with the model poses where the pixel coordinates are the same as those of the hash database. For our testing we coded four target classes, the M60, M113, HMMWV and BMP using an X-Y projection extraction (discarding depth, or Z, information) from the 3D Hash Point data sets given in Appendix B.

Any number of C-80 systems may be combined to increase the number of target classes or features that are processed in parallel. Iterative projection is 'scaleable' in the sense that the projections are multiple 2-D independent databases. Each projection can be coded and searched in parallel and attributed label feature sets likewise can be coded and searched. Additionally, algorithms to improve recognition can be implemented directly on the PPs, while the iterative process can be orchestrated by the MP.

The model sets that we used consisted of 12 poses (30° rotations) for each class that generated hash databases of roughly 300Kbytes each. These were processed into database images as described above. We presented an unknown M60 to the system with the following result:

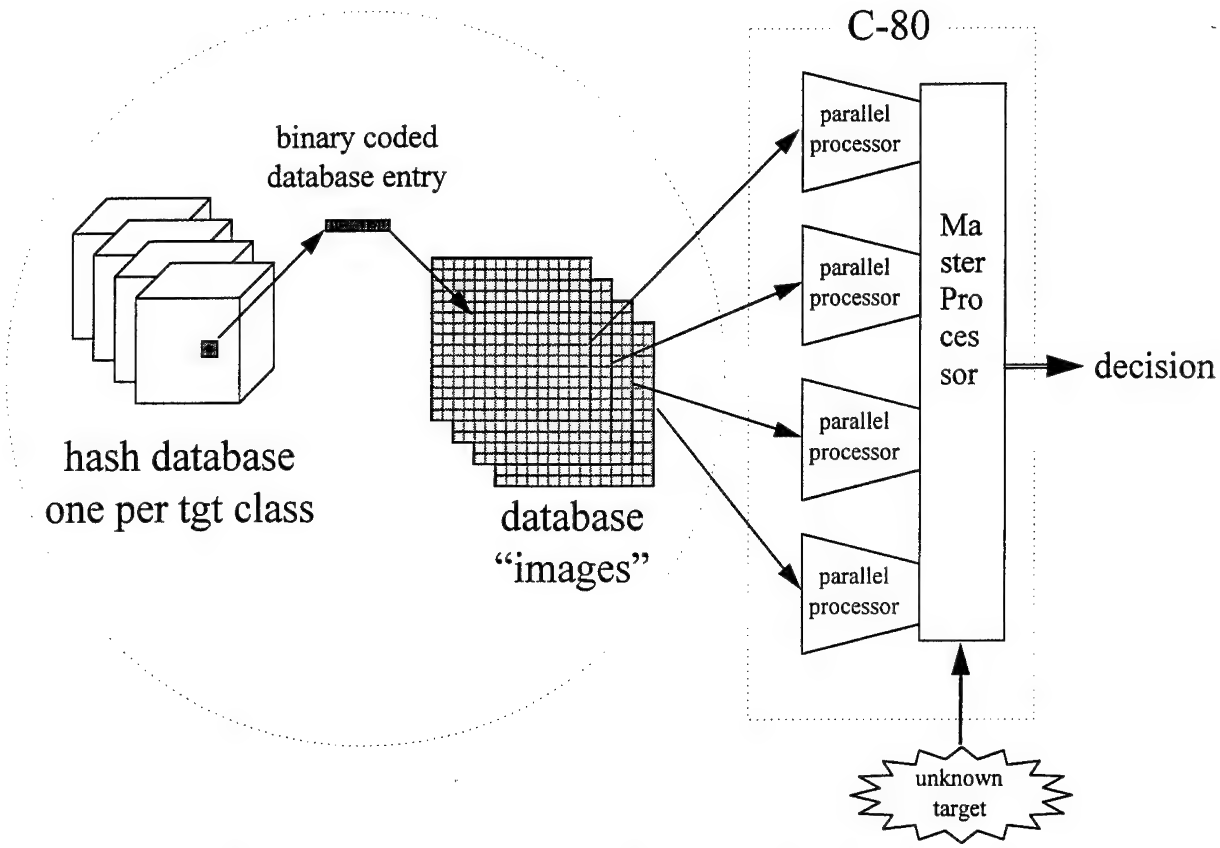


Figure 14. C-80 (TI MVP) Processing of Hash Database Images

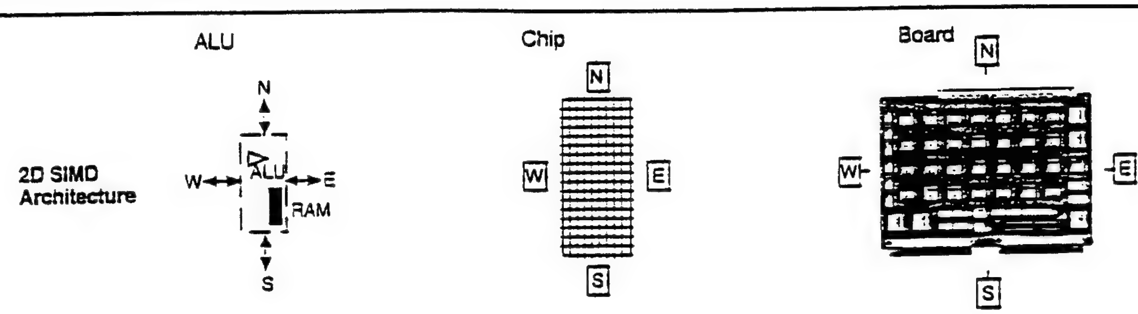
Table 3. C80 2D Hashing Results for an Unknown M60 Target

Processor/Class	Pose	Vote
HMMWV	60	7
M35	120	12
M60	**270**	**27**
M113	90	6

These results are significant in that the hashing occurred in parallel on a compact system capable of being fielded in military applications. Another point of significance is that the hashing database was processed as a novel *database image*. Images of target classes can be stored into Read-Only Memories (ROM) in multiple C-80 systems so that large numbers of classes and poses can be searched in parallel. As new target classes are developed or databases are improved the system can be easily field upgraded.

5. GAPP/PAL IMPLEMENTATION

The I-MATH/NYU team has the overall objective of implementing geometric hashing algorithms onto a commercial off the shelf (COTS) image processor optimum for a wide variety of military and commercial applications. These algorithms will provide real-time execution of automatic target recognizer and information fusion applications which use high dimensionality information. The Lockheed Martin GAPP/PAL* and the TI C80 processors have been chosen because both have proven parallel image processing architectures, which is the predominant ND hashing data format. As shown by Figure 15, the GAPP is best at SIMD type problems, primarily inter-pixel tasks. As previously discussed in Section 4.2, the C80 is best suited to intra-pixel and databasing tasks. Each processor is capable of running all of the anticipated algorithms; however, the hybrid combination of the two is most likely to produce the most efficient real-time data processing.



- Massively parallel array of processors (scalable to over 64,000 processors)
- Fine-grained, two-dimensional SIMD array of processors directly mirrors the image data structure for maximum efficiency in data handling
- 1-bit serial processors extend the RISC concept to the limit for maximum hardware efficiency with algorithm operations from 1-bit to floating point

Figure 15. 2D SIMD Architecture's Match of Image Data Structure

Table 4 shows the historical progression of GAPP chips. The current GAPP chip has evolved as silicon technology has evolved, but the architecture has remained consistent. the SIMD organization remains the preferred, and demonstrably the best image processor. In addition to bit-level and multi-bit image processing, GAPP chips efficiently perform floating point operations even though they are composed of simple 1-bit, bit-serial processing cells.

* Lockheed Martin has developed a family of Geometric Arithmetic Parallel Processors (GAPP) which are SIMD devices using Parallel Algebraic Logic (PAL). The current commercially available system is a GAPP-4, which is part of the PAL-1 family.

Table 4. CISP Architecture is based on mature GAPP Chips (geometric arithmetic parallel processor)

Chip Parameter	GAPP I (1983)	GAPP II (1984)	GAPP III (1990)	GAPP IV (1993)
Processing array elements	3 x 6	6 x 12	8 x 16	16 x 12
Clock Rate	5 MHz	15 MHz	25 MHz	40 MHz
Memory/Element	128 bits	128 bits	128 bits	192 bits
New features	-----	Global OR, fork	Segmented RAM enhanced ALU	3-port RAM, pipelining, enhanced ALU, JTAG
Foundry	NCR	NCR	LSI Logic/ORBIT	LSI Logic
Number of I/O pins	68	76	128	180
Technology (CMOS)	5μm	2μm	1.0μm	0.6μm
Power	0.8W	0.6W	0.85W	1.1W
1-bit edge ops/IC	23 MOPS	270 MOPS	914 MOPS	2560 MOPS
8-bit image ops/IC	4 MOPS	43 MOPS	178 MOPS	768 MOPS
Floating point ops/IC	0.4 MFLOPS	1 MFLOPS	4 MFLOPS	19 MFLOPS

The PAL I architecture is based on the fourth GAPP generation of fine-grained, massively parallel, two-dimensional grid processors developed by Lockheed Martin. The PAL I architecture may be configured for either embedded or workstation applications, although the initial PAL emphasis is on the commercial workstation accelerator configuration. The workstation configuration is useful as an inexpensive host for exploring new algorithm ideas and rapidly developing prototype demonstrations for new applications and customers. The 6uVME board format has been selected for the PAL I workstation because of its status as the predominant commercial packaging standard. The PAL I workstation speeds the execution of compute intensive image processing algorithms by several hundred times over the standard SPARC 20 host, thereby greatly accelerating the algorithm concept development and evaluation process. The PAL workstation development environment is supported with a complete set of state-of-the-art algorithm and software development tools, including Image Algebra, Khoros, Ada, and C++.

The PAL I architecture has been laid out on SEM-E and other military custom packaging formats in addition to the commercial 6uVME board. The PAL I architecture can be scaled directly, simply by adding or subtracting PAL array boards. Each PAL I array board contains over 4000 processing elements organized in a 64x72 grid, and throughput support is linearly scaleable between 1- and 16-board configurations. Throughput also scales depending on the precision of operation required at each step of the algorithm suite. A typical commercial workstation configuration consists of two array boards interconnected as a 128x72 grid and the controller/Sun interface board and provides more than 40 billion operations per second algorithm throughput support. The interface between the PAL accelerator and Sun workstation consists of a 16-bit parallel bi-directional port connected to the Sun's S-Bus with a commercially available interface board. This interface provides a 28 MB/s burst transfer rate interface that is adequate for video rate image transfers from Sun memory. The embedded PAL I configuration, based on 6u VME boards, uses a different interface and control board capable of multisensor and higher data rate I/O operation (4 asynchronous video streams allowing a total of 50 million 16-bit pixels

per second). The embedded I/O-controller board also contains two C44 processors in order to support stand-alone system operation.

The next-generation PAL II architecture has begun the detailed chip design phase in a contract with the LSI Logic foundry that will utilize their latest 0.35μm G10 process. The PAL II design will further exploit the unique characteristics and support the unique needs of image processing applications. Two major goals of the PAL II development include a further stretching of the scalability limits, both on the high and low ends, and a streamlining of the interfaceability of PAL II chips. Table 5 depicts the performance and scalability goals of the PAL II architecture. PAL II will be scaleable from a single 32x32 array chip to a full 512x512 array, thus providing a range of more than 250x in terms of throughput (i.e., from 10 GOPS to over 2.5 TOPS) performance. In its minimum configuration, a PAL II processor providing over 10 GOPS throughput support, over 30 GB/s computational memory bandwidth, and over 640 MB/s of I/O bandwidth is expected to cost less than $1000. On the other end of scalability, a 512x512 configuration will provide over 2.6 TOPS of throughput, over 7 TB/s of computational memory bandwidth, and over 5 GB/s of I/O bandwidth.

Table 5. PAL II Performance Objectives

	PAL II Array Size		
	128x128	**256x256**	**512x512**
Number of boards	1	4	16
Board area	25 sq in	100 sq in	400 sq in
Volume (assumes 1/2" pitch)	12.5 cu in	50 cu in	200 cu in
Weight	4 oz	1 lb	4 lb
Power	35 watts	140 watts	560 watts
System Cost (Qty 5000 boards)	$4K	$16K	$64K
Typical application	Expendable Munitions	Fire Control Systems	Supercomputer Workstation
Max external I/O BW	1.28 GB/s	2.56 GB/s	5.12 GB/s
Throughput: 1-bit edge ops 8-bit image ops 32-bit floating ops	 1311 GOPS 164 GOPS 1.8 GOPS	 5243 GOPS 655 GOPS 7.1 GOPS	 20.972 GOPS 2621 GOPS 28.6 GOPS
Array Computational Memory I/O bandwidth	492 GB/s	1966 GB/s	7864 GB/s
Price-performance (8-bit "image" operations	$0.024/MOPS $24/GOPS	$0.024/MOP $24/GOP	$0.024/MOP $24/GOP

APPENDIX A.

Generalized ND Hash Code

ND-Hash Directory Structure
==========================

```
  ndhash
    doc         - Documentation files.
    src         - Source (.h, .c) files, makefile,
                  .o files, executable "ndhash".
                - Directory "out" - has results files *.out,
                  summary of results file "results.txt".
    targets     - Model and test target features file "targets.txt".
```

How to Run ND-Hash
==================

(1) cd ndhash/src

(2) ndhash ../targets/targets.txt hum_cargo 0 30 7 14 18

(This command runs ndhash, reading the model and test target feature points from the file ../targets/targets.txt, using target(type=hum_cargo, poseX=0, poseZ=30) as the test target, and using feature point triple (7,14,18) of the test target as the test target's basis. Of course, any target in the target features file (in this case ../targets/targets.txt) can be used as the test target.)

(ndhash takes an optional last argument that causes random uniform noise to be added to each coordinate of the test target's feature points. This optional last argument defaults to 0 (e.g., in the above example).
For example:

```
ndhash ../targets/targets.txt hum_cargo 0 30 7 14 18 10
```

runs ndhash exactly as in the above example, but -10..10 units of random uniform noise is added to each coordinate of the test target's feature points.)

(ndhash prints out the results of its computations, so you may want to redirect its output to a file; e.g.:

```
ndhash ../targets/targets.txt hum_cargo 0 30 7 14 18 >
hum_cargo.0.30.out
```

The top 30 model, basis pairs that best match the test target are printed out at the end in sorted order.)

```
##
## makefile for ndhash
##

COMPILER=gcc

all:                    ndhash

ndhash:                 utils.o matrix.o hashtbl.o recog.o
                                        $(COMPILER) -o ndhash utils.o matrix.o hashtbl.o
recog.o -lm

utils.o:                utils.h utils.c
                                        $(COMPILER) -c utils.c

matrix.o:               matrix.h matrix.c utils.h
                                        $(COMPILER) -c matrix.c

hashtbl.o:              hashtbl.h hashtbl.c utils.h matrix.h
                                        $(COMPILER) -c hashtbl.c

recog.o:                recog.c utils.h matrix.h hashtbl.h
                                        $(COMPILER) -c recog.c

clean:
                              rm -f *.o
                              rm -f ndhash
```

```
/*
 utils.c
*/

#include "utils.h"

int _MsgVerbosity_ = 0;

extern void Assert(boolean b, CharPtr msg)
{
 if (!b) {
   printf("Assert(): Failure: %s\n", msg);
   exit(1);
 }
}

/*
 hashtbl.c
*/
```

```
#include "utils.h"
#include "hashtbl.h"

#define _BackgroundDistr_ 0.00001

/*
 Point
*/

extern Point New_Point(int dimension)
{
 Point point = (Point)malloc(dimension*sizeof(DimensionType));
 return point;
}

extern void Destroy_Point(Point point)
{
 free(point);
}

extern void Copy_Point(int dimension, Point p1, Point p2)
{
 int i;

 for (i = 0; i < dimension; i++) {
  p2[i] = p1[i];
 }
}

extern DimensionType Get_Distance_Point_Point
 (int dimension, Point p1, Point p2)
{
 DimensionType d = 0;
 int i;

 for (i = 0; i < dimension; i++) {
  DimensionType t1;

  t1 = p1[i]-p2[i];
  d += t1*t1;
 }
 d = sqrt(d);
 return d;
}

extern void Add_Point_Point(int dimension, Point p1, Point p2, Point p3)
{
 int i;

 for (i = 0; i < dimension; i++) {
  p3[i] = p1[i] + p2[i];
```

```
  }
}

extern void Subtract_Point_Point(int dimension, Point p1, Point p2, Point p3)
{
  int i;

  for (i = 0; i < dimension; i++) {
    p3[i] = p1[i] - p2[i];
  }
}

/*
  Basis
*/

extern Basis New_Basis(int dimension)
{
  Basis basis = (Basis)malloc(dimension*sizeof(FeatureNum));
  return basis;
}

extern void Destroy_Basis(Basis basis)
{
  free(basis);
}

extern void Copy_Basis(int dimension, Basis b1, Basis b2)
{
  int i;

  for (i = 0; i < dimension; i++) {
    b2[i] = b1[i];
  }
}
/*
  HashTableEntry
*/

extern HashTableEntryPtr New_HashTableEntry()
{
  HashTableEntryPtr ep =
    (HashTableEntryPtr)malloc(sizeof(HashTableEntry));
  return ep;
}

extern HashTableEntryPtr New_Set_HashTableEntry
  (
    int dimension,
    int basisDimension,
    Point point,
```

```
  ModelNum modelNum,
  Basis basis,
  PDFPtr pdfPtr,
  StatsPtr statsPtr,
  FeatureType featureType
 )
{

  HashTableEntryPtr ep;
  Point       entryPoint;
  ModelNum    entryModelNum;
  Basis       entryBasis;
  PDFPtr      entryPdfPtr;
  StatsPtr    entryStatsPtr;
  FeatureType entryFeatureType;

  ep = New_HashTableEntry();

  entryPoint = New_Point(dimension);
  Copy_Point(dimension, point, entryPoint);
  Set_HashTableEntryPtr_Point(ep, entryPoint);

  entryModelNum = modelNum;
  Set_HashTableEntryPtr_ModelNum(ep, entryModelNum);

  entryBasis = New_Basis(basisDimension);
  Copy_Basis(basisDimension, basis, entryBasis);
  Set_HashTableEntryPtr_Basis(ep, entryBasis);

  entryPdfPtr = pdfPtr;
  Set_HashTableEntryPtr_PdfPtr(ep, entryPdfPtr);

  entryStatsPtr = statsPtr;
  Set_HashTableEntryPtr_StatsPtr(ep, entryStatsPtr);

  entryFeatureType = featureType;
  Set_HashTableEntryPtr_FeatureType(ep, entryFeatureType);

  Set_HashTableEntryPtr_Vote(ep, _VoteNull_);

  return ep;

}

/*
 HashTableEntryNode
*/

extern HashTableEntryNodePtr New_HashTableEntryNode()
{
 HashTableEntryNodePtr np =
```

```
  (HashTableEntryNodePtr)malloc(sizeof(HashTableEntryNode));
 Set_HashTableEntryNodePtr_EntryPtr(np, NULL);
 Set_HashTableEntryNodePtr_Distance(np, 0);
/*
 Set_HashTableEntryNodePtr_Vote(np, _VoteNull_);
*/
 Set_HashTableEntryNodePtr_NextPtr(np, NULL);
 return np;
}

/*
 HashTableEntryList
*/

extern HashTableEntryListPtr New_HashTableEntryList()
{
 HashTableEntryListPtr lp =
  (HashTableEntryListPtr)malloc(sizeof(HashTableEntryList));
 Set_HashTableEntryListPtr_FirstNodePtr(lp, NULL);
 Set_HashTableEntryListPtr_LastNodePtr(lp, NULL);
 Set_HashTableEntryListPtr_VotesHaveBeenComputed(lp, FALSE);
 return lp;
}
extern HashTableEntryListPtr InsertHead_HashTableEntryListPtr_EntryPtr_Distance
 (HashTableEntryListPtr lp, HashTableEntryPtr ep, DimensionType distance)
{
 HashTableEntryNodePtr np;
 HashTableEntryNodePtr fnp;
 HashTableEntryNodePtr lnp;

 np = New_HashTableEntryNode();
 Set_HashTableEntryNodePtr_EntryPtr(np, ep);
 Set_HashTableEntryNodePtr_Distance(np, distance);
 fnp = Get_HashTableEntryListPtr_FirstNodePtr(lp);
 Set_HashTableEntryNodePtr_NextPtr(np, fnp);
 Set_HashTableEntryListPtr_FirstNodePtr(lp, np);
 lnp = Get_HashTableEntryListPtr_LastNodePtr(lp);
 if (lnp == NULL) {
  Set_HashTableEntryListPtr_LastNodePtr(lp, np);
 }
 return lp;
}
```

```
/*
 HashTable Creation and Access Functions
*/

extern HashTablePtr New_HashTable
 (
```

```
  int dimension2,
  DimensionType *dimensionMinVals2,
  DimensionType *dimensionMaxVals2,
  int *dimensionNumPartitions2
 )
{
 HashTablePtr htp;
 DimensionType *dimMinVals;
 DimensionType *dimMaxVals;
 int *dimNumPartitions;
 DimensionType *dimPartitionSizes;
 int numB;
 int i;
 HashTableBucketPtr *bps;

 htp =
  (HashTablePtr)malloc(sizeof(HashTable));
 Set_HashTablePtr_Dimension(htp, dimension2);
 dimMinVals =
  (DimensionType *)malloc(dimension2*sizeof(DimensionType));
 dimMaxVals =
  (DimensionType *)malloc(dimension2*sizeof(DimensionType));
 dimNumPartitions = (int *)malloc(dimension2*sizeof(int));
 dimPartitionSizes =
  (DimensionType *)malloc(dimension2*sizeof(DimensionType));
 Set_HashTablePtr_DimensionMinVals(htp, dimMinVals);
 Set_HashTablePtr_DimensionMaxVals(htp, dimMaxVals);
 Set_HashTablePtr_DimensionNumPartitions(htp, dimNumPartitions);
 Set_HashTablePtr_DimensionPartitionSizes(htp, dimPartitionSizes);
 numB = 1;
 for (i = 0; i < dimension2; i++) {
  DimensionType dimensionMinVals2I;
  DimensionType dimensionMaxVals2I;
  int dimensionNumPartitions2I;
  DimensionType dimPartitionSizesI;

  dimensionMinVals2I = dimensionMinVals2[i];
  dimensionMaxVals2I = dimensionMaxVals2[i];
  Set_HashTablePtr_DimensionMinValI(htp, i, dimensionMinVals2I);
  Set_HashTablePtr_DimensionMaxValI(htp, i, dimensionMaxVals2I);
  dimensionNumPartitions2I = dimensionNumPartitions2[i];
  Set_HashTablePtr_DimensionNumPartitionI(htp, i, dimensionNumPartitions2I);
  numB *= dimensionNumPartitions2I;
  dimPartitionSizesI =
   (dimensionMaxVals2I - dimensionMinVals2I + 1) / dimensionNumPartitions2I;
  Set_HashTablePtr_DimensionPartitionSizeI(htp, i, dimPartitionSizesI);
 }
 Set_HashTablePtr_NumBuckets(htp, numB);
 bps =
  (HashTableBucketPtr *)malloc(numB*sizeof(HashTableBucketPtr));
 Set_HashTablePtr_BucketPtrs(htp, bps);
```

```
  for (i = 0; i < numB; i++) {
    HashTableEntryListPtr lp = New_HashTableEntryList();
    HashTableBucketPtr bp = lp;
    Set_HashTablePtr_BucketPtrI(htp, i, bp);
  }
  return htp;
}

extern int Get_HashTablePtr_Point_BucketNum_BucketMidpoint
  (HashTablePtr htp, Point point, Point bucketMidpoint)
{
  int dimension;
  int dimensionMinus1;
  int partitionNum;
  int i;

  dimension = Get_HashTablePtr_Dimension(htp);
  dimensionMinus1 = dimension-1;
  partitionNum = 0;
  for (i = dimensionMinus1; i >= 0; i--) {
    int partitionNumI;
    int prevDimsNumPartitions;

    partitionNumI =
      (int)
      (
        (point[i] - Get_HashTablePtr_DimensionMinValI(htp, i))
          / Get_HashTablePtr_DimensionPartitionSizeI(htp, i)
      );
/*
printf("debug1: 10: partitionNumI=%d\n", partitionNumI); fflush(stdout);
  printf("debug1: 15: point[i]=%.2f, %.2f, %.2f\n",
    point[i],
    Get_HashTablePtr_DimensionMinValI(htp, i),
    Get_HashTablePtr_DimensionPartitionSizeI(htp, i)
    );
  fflush(stdout);
*/
    bucketMidpoint[i] =
      Get_HashTablePtr_DimensionMinValI(htp, i)
        + partitionNumI*Get_HashTablePtr_DimensionPartitionSizeI(htp, i)
        + Get_HashTablePtr_DimensionPartitionSizeI(htp, i)/2;
/*
printf("debug1: 20: bucketMidpointI=%.2f\n", bucketMidpoint[i]); fflush(stdout);
*/
    if (i == dimensionMinus1) {
      prevDimsNumPartitions = 1;
    } else {
      prevDimsNumPartitions *=
        Get_HashTablePtr_DimensionNumPartitionI(htp, i+1);
    }
```

```
/*
printf("debug1: 30: prevDimsNumPartitions=%d\n", prevDimsNumPartitions); fflush(stdout);
*/
  partitionNum += partitionNumI*prevDimsNumPartitions;
/*
printf("debug1: 40: partitionNum=%d\n", partitionNum); fflush(stdout);
*/
 }
 return partitionNum;
}

extern void InsertIntoHypercube_HashTablePtr_EntryPtr
 (
  HashTablePtr htp, HashTableEntryPtr ep,
  int partitionRangeMinIndex, int partitionRangeMaxIndex,
  Point entryPoint, int dimension, Point point, Point bucketMidpoint,
  int loopDimensionNum
 )
{
 if (loopDimensionNum == dimension) {
  /* base case */

  int bucketNum;
  DimensionType distance;
  HashTableBucketPtr bp;
  HashTableEntryListPtr lp;

  bucketNum =
   Get_HashTablePtr_Point_BucketNum_BucketMidpoint
    (htp, point, bucketMidpoint);
  distance =
   Get_Distance_Point_Point(dimension, point, bucketMidpoint);
  bp = Get_HashTablePtr_BucketPtrI(htp, bucketNum);
  lp = (HashTableEntryListPtr)bp;
  InsertHead_HashTableEntryListPtr_EntryPtr_Distance(lp, ep, distance);

if (5 <= _MsgVerbosity_) {
 printf("Inserted point=(%d,%d,%d), modelNum=%d, basis=(%d,%d,%d), distance=%.2f into bucket
%d\n",
  (int)Get_HashTableEntryPtr_Point(ep)[0],
  (int)Get_HashTableEntryPtr_Point(ep)[1],
  (int)Get_HashTableEntryPtr_Point(ep)[2],
  Get_HashTableEntryPtr_ModelNum(ep),
  Get_HashTableEntryPtr_Basis(ep)[0],
  Get_HashTableEntryPtr_Basis(ep)[1],
  Get_HashTableEntryPtr_Basis(ep)[2],
  distance,
  bucketNum
  );
 fflush(stdout);
}
```

```
  } else {
   /* recursive case */

   DimensionType partitionSizeD;
   int p;

   partitionSizeD =
     Get_HashTablePtr_DimensionPartitionSizeI(htp, loopDimensionNum);
   for (p = partitionRangeMinIndex; p <= partitionRangeMaxIndex; p++) {
     point[loopDimensionNum] = entryPoint[loopDimensionNum] + p*partitionSizeD;
     InsertIntoHypercube_HashTablePtr_EntryPtr
       (
         htp, ep,
         partitionRangeMinIndex, partitionRangeMaxIndex,
         entryPoint, dimension, point, bucketMidpoint,
         loopDimensionNum+1
       );
   }
  }
}

extern HashTablePtr Insert_HashTablePtr_EntryPtr
  (
   HashTablePtr htp, HashTableEntryPtr ep,
   int partitionRangeMinIndex, int partitionRangeMaxIndex
  )
{
  Point entryPoint = Get_HashTableEntryPtr_Point(ep);
  int dimension = Get_HashTablePtr_Dimension(htp);
  Point point = New_Point(dimension);
  Point bucketMidpoint = New_Point(dimension);

  Copy_Point(dimension, entryPoint, point);

  InsertIntoHypercube_HashTablePtr_EntryPtr
   (
     htp, ep,
     partitionRangeMinIndex, partitionRangeMaxIndex,
     entryPoint, dimension, point, bucketMidpoint,
     0
   );

  Destroy_Point(point);
  Destroy_Point(bucketMidpoint);

  return htp;
}

extern HashTableEntryListPtr GetHashTableEntryListForEntry
  (HashTablePtr htp, HashTableEntryPtr ep)
{
```

```
  int dimension = Get_HashTablePtr_Dimension(htp);

  Point entryPoint = Get_HashTableEntryPtr_Point(ep);
  Point entryBucketMidpoint = New_Point(dimension);

  int entryBucketNum;
  DimensionType entryDistance;
  HashTableBucketPtr entryBp;
  HashTableEntryListPtr entryLp;

  entryBucketNum =
   Get_HashTablePtr_Point_BucketNum_BucketMidpoint
    (htp, entryPoint, entryBucketMidpoint);
  entryDistance =
   Get_Distance_Point_Point(dimension, entryPoint, entryBucketMidpoint);
  entryBp = Get_HashTablePtr_BucketPtrI(htp, entryBucketNum);
  entryLp = (HashTableEntryListPtr)entryBp;

  Destroy_Point(entryBucketMidpoint);

  return entryLp;

}

/*
 Compute Vote Functions
*/

extern void ComputeVotesInHashTableForEntry
 (
  HashTablePtr htp, HashTableEntryPtr ep,
  int partitionRangeMinIndex, int partitionRangeMaxIndex
 )
{

 Point entryPoint = Get_HashTableEntryPtr_Point(ep);
 int dimension = Get_HashTablePtr_Dimension(htp);
 Point point = New_Point(dimension);
 Point bucketMidpoint = New_Point(dimension);

 Copy_Point(dimension, entryPoint, point);

 ComputeVotesInHypercubeForEntry
  (
   htp, ep,
   partitionRangeMinIndex, partitionRangeMaxIndex,
   entryPoint, dimension, point, bucketMidpoint,
   0
  );
```

```
 Destroy_Point(point);
 Destroy_Point(bucketMidpoint);

}

extern void ComputeVotesInHypercubeForEntry
 (
  HashTablePtr htp, HashTableEntryPtr ep,
  int partitionRangeMinIndex, int partitionRangeMaxIndex,
  Point entryPoint, int dimension, Point point, Point bucketMidpoint,
  int loopDimensionNum
 )
{
 if (loopDimensionNum == dimension) {
  /* base case */

  int bucketNum;
  HashTableBucketPtr bp;
  HashTableEntryListPtr lp;

  bucketNum =
   Get_HashTablePtr_Point_BucketNum_BucketMidpoint
    (htp, point, bucketMidpoint);
  bp = Get_HashTablePtr_BucketPtrI(htp, bucketNum);
  lp = (HashTableEntryListPtr)bp;
  ComputeVotesInHashTableEntryListForEntry
   (htp, lp, ep);

 } else {
  /* recursive case */

  DimensionType partitionSizeD;
  int p;
  partitionSizeD =
   Get_HashTablePtr_DimensionPartitionSizeI(htp, loopDimensionNum);
  for (p = partitionRangeMinIndex; p <= partitionRangeMaxIndex; p++) {
   point[loopDimensionNum] = entryPoint[loopDimensionNum] + p*partitionSizeD;
   ComputeVotesInHypercubeForEntry
    (
     htp, ep,
     partitionRangeMinIndex, partitionRangeMaxIndex,
     entryPoint, dimension, point, bucketMidpoint,
     loopDimensionNum+1
    );
  }
 }
}

extern void ComputeVotesInHashTableEntryListForEntry
 (HashTablePtr htp, HashTableEntryListPtr lp, HashTableEntryPtr ep)
{
```

```
 int dimension = Get_HashTablePtr_Dimension(htp);

 Point epPoint = Get_HashTableEntryPtr_Point(ep);

 HashTableEntryNodePtr lnp = Get_HashTableEntryListPtr_FirstNodePtr(lp);

if (0 <= _MsgVerbosity_) {
 printf("Computing votes in bin for test entry: point=(%.2f,%.2f,%.2f); basis=(%d,%d,%d) ...\n",
  Get_HashTableEntryPtr_Point(ep)[0],
  Get_HashTableEntryPtr_Point(ep)[1],
  Get_HashTableEntryPtr_Point(ep)[2],
  Get_HashTableEntryPtr_Basis(ep)[0],
  Get_HashTableEntryPtr_Basis(ep)[1],
  Get_HashTableEntryPtr_Basis(ep)[2]
  );
 fflush(stdout);
}

 if (lnp != NULL) {
  Set_HashTableEntryListPtr_VotesHaveBeenComputed(lp, TRUE);
 }

 while (lnp != NULL) {

  HashTableEntryPtr lep = Get_HashTableEntryNodePtr_EntryPtr(lnp);

  Point lepPoint = Get_HashTableEntryPtr_Point(lep);

  PDFPtr lepPdfPtr = Get_HashTableEntryPtr_PdfPtr(lep);

  Vote oldVote = Get_HashTableEntryNodePtr_Vote(lnp);

  Vote vote;

/*
  vote =
   Compute_DistanceVote_PredPoint_ExtrPoint
    (dimension, lepPoint, epPoint);
*/

  vote =
   Compute_Vote_PDFPtr_PredPoint_ExtrPoint_BackgroundDistrFuncPtr
    (lepPdfPtr, lepPoint, epPoint, &Compute_BackgroundDistr);

  Assert( (vote == _VoteNull_) || (vote >= _VoteMin_),
   "ComputeVotesInHashTableEntryListForEntry(): non-null vote < _VoteMin_");

  if (
    (vote != _VoteNull_) /* this node's new vote is for a match */
     &&
```

```
      (vote > oldVote) /* this node's new vote is better than its old vote */
     ) {

     Set_HashTableEntryNodePtr_Vote(lnp, vote);

if (3 <= _MsgVerbosity_) {
 printf("Hash table entry: point=(%d,%d,%d); modelNum=%d; basis=(%d,%d,%d); vote=%.2f\n",
  (int)Get_HashTableEntryPtr_Point(lep)[0],
  (int)Get_HashTableEntryPtr_Point(lep)[1],
  (int)Get_HashTableEntryPtr_Point(lep)[2],
  Get_HashTableEntryPtr_ModelNum(lep),
  Get_HashTableEntryPtr_Basis(lep)[0],
  Get_HashTableEntryPtr_Basis(lep)[1],
  Get_HashTableEntryPtr_Basis(lep)[2],
  vote
  );
 fflush(stdout);
}

  }
  lnp = Get_HashTableEntryNodePtr_NextPtr(lnp);

 }

}

extern Vote Compute_DistanceVote_PredPoint_ExtrPoint
 (
  int dimension, Point predictedPoint, Point extractedPoint
 )
{

 DimensionType distance =
  Get_Distance_Point_Point(dimension, predictedPoint, extractedPoint);

 Vote vote = ceil(sqrt(10*10*10)) - distance;

 return vote;

}

extern float Compute_Log_PDFPtr_PredPoint_ExtrPoint
 (PDFPtr pdfp, Point predictedPoint, Point extractedPoint)
{
 int dimension = Get_PDFPtr_Dimension(pdfp);
 Point mean = Get_PDFPtr_Mean(pdfp);
 Element detCov = Get_PDFPtr_DetCov(pdfp);
 Matrix invCov = Get_PDFPtr_InvCov(pdfp);

 float term1;
```

```
 Point extrMinusPred;
 Point extrMinusPredMinusMean;
 float term2;

 float logpdf;

 term1 = -0.5 * log( pow(2*PI, dimension) * detCov );
/*
printf("Compute_Log_...() term1=%.2f\n", term1); fflush(stdout);
*/

 extrMinusPred = New_Point(dimension);
 Subtract_Point_Point
  (dimension, extractedPoint, predictedPoint, extrMinusPred);
 extrMinusPredMinusMean = New_Point(dimension);
 Subtract_Point_Point
  (dimension, extrMinusPred, mean, extrMinusPredMinusMean);
 if (dimension == 3) {
  Point product1 = New_Point(dimension);
  Element product2;
  Mult_Vector_Matrix
   (extrMinusPredMinusMean, dimension, invCov, dimension, dimension, product1);
/*
printf("Compute_Log_...() product1 = (%.2f,%.2f,%.2f)\n", product1[0], product1[1], product1[2]);
fflush(stdout);
*/
  Mult_Vector_Vector(product1, extrMinusPredMinusMean, dimension, &product2);
/*
printf("Compute_Log_...() product2=%.2f\n", product2); fflush(stdout);
*/
  term2 = -0.5 * product2;
/*
printf("Compute_Log_...() term2=%.2f\n", term2); fflush(stdout);
*/
 } else {
  printf("Compute_Log_PDFPtr_PredPoint_ExtrPoint(): dimension != 3; exiting.\n");
  exit(1);
 }

 logpdf = term1 + term2;

 return logpdf;
}

extern float Compute_BackgroundDistr(Point extractedPoint)
{
 return _BackgroundDistr_;
}

#define _InitialVoteMatchThreshold_ -2
```

```
extern Vote Compute_Vote_PDFPtr_PredPoint_ExtrPoint_BackgroundDistrFuncPtr
 (
   PDFPtr pdfp, Point predictedPoint, Point extractedPoint,
   BackgroundDistrFuncPtr backgroundDistrFuncPtr
 )
{

 float term1;
 float term2;
 float vote;

/*
printf("Compute_Vote_...() model entry: point=(%.2f,%.2f,%.2f)\n",
 predictedPoint[0],
 predictedPoint[1],
 predictedPoint[2]
 );
fflush(stdout);
*/

 term1 =
   Compute_Log_PDFPtr_PredPoint_ExtrPoint
    (pdfp, predictedPoint, extractedPoint);
/*
printf("Compute_Vote_...() term1=%.2f\n", term1); fflush(stdout);
*/

 term2 = -log( (*backgroundDistrFuncPtr)(extractedPoint) );
/*
printf("Compute_Vote_...() term2=%.2f\n", term2); fflush(stdout);
*/

 vote = term1 + term2;
/*
printf("Compute_Vote_...() initial vote=%.2f\n", vote); fflush(stdout);
*/

 if (vote < _InitialVoteMatchThreshold_) {
   /* model entry doesn't match test entry */
   vote = _VoteNull_;
 } else {
   /* model entry matches test entry */
   vote += (_VoteMin_ - _InitialVoteMatchThreshold_);
 }

 return vote;

}

extern void PrintMsg(int msgVerbosityLevel, CharPtr msg)
{
```

```
  if (msgVerbosityLevel <= _MsgVerbosity_) {
    printf("%s", msg);
    fflush(stdout);
  }
}
```

```
/*
  recog.c

  This module implements the recognizer.
*/

#include "utils.h"
#include "hashtbl.h"

/*
  Hash space dimensions: x, y, range.
*/

#define _Dimension_ 3

static DimensionType _DimensionMinVals[_Dimension_] =
  { -500, -500, -300 };

static DimensionType _DimensionMaxVals[_Dimension_] =
  {  499,  499,  299 };

static int _DimensionNumPartitions[_Dimension_] =
  {  100,  100,   60 };

/*
  Basis dimension.
*/

/*
  We'll do 3D translation+rotation+scale invariance, so we need
  3 points in a basis.
*/
#define _BasisDimension_ 3

/*
  ModelHashTable
*/

static HashTablePtr _ModelHashTablePtr;

/*
  Name of file that has the feature points of all of the model and test targets.
*/
```

```
static String _TargetsFileName;

/*
 PDF of model entries.
*/

static PDF _ModelEntryPdf;

/*
 Model specification.
*/

#define _ModelDegreeIncrement_ 30
#define _NumModelsPerModelType_ (360/_ModelDegreeIncrement_)
#define _NumModelTypes_ 4
#define _NumModels_ (_NumModelTypes_ * _NumModelsPerModelType_)

static CharPtr _ModelTypes[_NumModelTypes_] =
  {
   "hum_cargo",
   "m113",
   "m35_canvas",
   "m60"
  };

#define _MaxNumBasesDim0_ 50
#define _MaxNumBasesDim1_ 50
#define _MaxNumBasesDim2_ 50

static ModelNum _ModelNum = 0;

static int NumbersOfModelPointsInModel
  [_NumModels_]; /* model number */

/*

  For now, we'll set these to 0, which will cause a hash table entry
  to be inserted into only the bin in which it lands, and not into
  neighboring bins.

  Strike the preceding paragraph.
  We'll insert an entry into a hypercube of bins, where the hypercube
  is centered at the bin in which the entry lands, and the size of the
  hypercube in each dimension is
  (_InsertEntryPartitionRangeMaxIndex - _InsertEntryPartitionRangeMinIndex + 1)
  bins.

  Strike the preceding paragraph.
  We'll go back to using 0 because otherwise, multiple occurrences of a
  model entry can vote multiple times, which isn't the desired behavior.
```

```
*/
static int _InsertEntryPartitionRangeMinIndex = 0;
static int _InsertEntryPartitionRangeMaxIndex = 0;

/*
 These are similar to _InsertEntryPartitionRange...Index.
*/
static int _ComputeVotePartitionRangeMinIndex = -1;
static int _ComputeVotePartitionRangeMaxIndex = 1;

/*
 Array ModelTypesPoses maps (modelNumber) to (modelType, poseX, poseZ).
*/

typedef struct ModelTypePose
 {
  String modelType;
  String poseX;
  String poseZ;
 } ModelTypePose;

static ModelTypePose ModelTypesPoses[_NumModels_];

/*
 Array Votes maps (modelNumber, basis) to (Vote).
*/

static Vote Votes
 [_NumModels_]   /* model number */
 [_MaxNumBasesDim0_] /* basis feature number 0 */
 [_MaxNumBasesDim1_] /* basis feature number 1 */
 [_MaxNumBasesDim2_] /* basis feature number 2 */
 ;

static int NumbersOfModelEntriesThatContributedToVote
 [_NumModels_]   /* model number */
 [_MaxNumBasesDim0_] /* basis feature number 0 */
 [_MaxNumBasesDim1_] /* basis feature number 1 */
 [_MaxNumBasesDim2_] /* basis feature number 2 */
 ;

#define _VotePenaltyForUnmatchedModelPoint_ (3.0)

/*
 Array ModelNumBasisVoteArray maps (int) to (ModelNumBasisVote).
*/

typedef struct ModelNumBasisVote
 {
  ModelNum modelNum;
  FeatureNum basisFeatureNum0;
```

```
    FeatureNum basisFeatureNum1;
    FeatureNum basisFeatureNum2;
    Vote vote;
  } ModelNumBasisVote;

#define ModelNumBasisVoteArraySize \
  (_NumModels_ * _MaxNumBasesDim0_*_MaxNumBasesDim1_*_MaxNumBasesDim2_)

static ModelNumBasisVote ModelNumBasisVoteArray[ ModelNumBasisVoteArraySize ];

#define _NumWinningModelBasisPairs_ 30

/*
  Test target.
*/

static String _TestTargetType;
static String _TestTargetPoseX;
static String _TestTargetPoseZ;
static String _TestTargetBasisFeatureNum0;
static String _TestTargetBasisFeatureNum1;
static String _TestTargetBasisFeatureNum2;
static String _TestTargetPlusMinusRandomUniformNoise;

/*
  TargetPoints

  Holds the feature points of a target.
*/
#define _MaxNumTargetPoints_ 50
typedef struct TargetPoints
  {
    Point points[_MaxNumTargetPoints_];
    int numPoints;
  } TargetPoints;

typedef TargetPoints *TargetPointsPtr;

static void Init_TargetPoints(TargetPointsPtr tpp)
{
  int i;

  for (i = 0; i < _MaxNumTargetPoints_; i++) {
    tpp->points[i] = New_Point(_Dimension_);
  }
  tpp->numPoints = 0;
}

static void UnInit_TargetPoints(TargetPointsPtr tpp)
{
  int i;
```

```
  for (i = 0; i < _MaxNumTargetPoints_; i++) {
    Destroy_Point(tpp->points[i]);
  }
  tpp->numPoints = 0;
}

static void Print_TargetPoints(int tpVerbosity, TargetPointsPtr tpp)
{
  int i;

  if (tpVerbosity <= _MsgVerbosity_) {
    for (i = 0; i < tpp->numPoints; i++) {
      printf("%8.2f\t%8.2f\t%8.2f\n",
        tpp->points[i][0],
        tpp->points[i][1],
        tpp->points[i][2]
        );
    }
    fflush(stdout);
  }
}



/*
  Read target (targetType, targetPoseX, targetPoseZ)'s feature points
  from file targetsFileName and store them in TargetPoints *tpp.
*/
static void ReadTargetPointsFromTargetsFile
  (
    TargetPointsPtr tpp,
    CharPtr targetType,
    CharPtr targetPoseX,
    CharPtr targetPoseZ,
    CharPtr targetsFileName,
    boolean targetIsModel
  )
{

  FILE *fp;
  int returnCode;
  String string;

  fp = fopen(targetsFileName, "r");
  Assert(fp != NULL, "ReadTargetPointsFromTargetsFile(): fopen()");

  strcpy(string, "");
  returnCode = fscanf(fp, "%s", string);
  while (returnCode != EOF) {
    if (!strcmp(string, targetType)) {
```

```
    String string1, string2, string3;

    returnCode = fscanf(fp, "%s", string1);
    returnCode = fscanf(fp, "%s", string2);
    returnCode = fscanf(fp, "%s", string3);
    if ( !strcmp(string2, targetPoseX) && !strcmp(string3, targetPoseZ) ) {
     String string4;

     /* Eat filename. */
     strcpy(string4, "");
     returnCode = fscanf(fp, "%s", string4);
if (4 <= _MsgVerbosity_) {
  printf("%s\n", string4);
  fflush(stdout);
}

     tpp->numPoints = 0;
     strcpy(string4, "");
     returnCode = fscanf(fp, "%s", string4);
     while ( (string4 != NULL) && strcmp(string4, "End") ) {
      String string5, string6;
      float targetX, targetY, targetRange;

      returnCode = fscanf(fp, "%s", string5);
      returnCode = fscanf(fp, "%s", string6);

      sscanf(string4, "%f", &targetX);
      sscanf(string5, "%f", &targetY);
      sscanf(string6, "%f", &targetRange);
if (4 <= _MsgVerbosity_) {
  printf("%d\t%d\t%d\n", (int)targetX, (int)targetY, (int)targetRange);
  fflush(stdout);
}
      tpp->points[tpp->numPoints][0] = targetX;
      tpp->points[tpp->numPoints][1] = targetY;
      tpp->points[tpp->numPoints][2] = targetRange;
      tpp->numPoints++;

      strcpy(string4, "");
      returnCode = fscanf(fp, "%s", string4);
     }
     break;
    }
   }

   strcpy(string, "");
   returnCode = fscanf(fp, "%s", string);
  }

  if (targetIsModel) {
   NumbersOfModelPointsInModel[_ModelNum] = tpp->numPoints;
```

```
  }

  returnCode = fclose(fp);
  Assert(returnCode != EOF, "ReadTargetPointsFromTargetsFile(): fclose()");

}

#define _RandomIntMod_ 256

static float GetPlusMinusRandomUniformNoise(float plusMinusRandomUniformNoise)
{
  int randomInt;
  float randomFloat;

  randomInt = rand() % _RandomIntMod_;
  randomFloat = ((float)randomInt)/((float)_RandomIntMod_);
  randomFloat *= (2*plusMinusRandomUniformNoise);
  randomFloat -= plusMinusRandomUniformNoise;
  return randomFloat;
}

static void AddPlusMinusRandomUniformNoiseToTargetPoints
  (TargetPointsPtr tpp, float plusMinusRandomUniformNoise)
{

  int i;

  for (i = 0; i < tpp->numPoints; i++) {
    float noise0 = GetPlusMinusRandomUniformNoise(plusMinusRandomUniformNoise);
    float noise1 = GetPlusMinusRandomUniformNoise(plusMinusRandomUniformNoise);
    float noise2 = GetPlusMinusRandomUniformNoise(plusMinusRandomUniformNoise);
    tpp->points[i][0] += noise0;
    tpp->points[i][1] += noise1;
    tpp->points[i][2] += noise2;
  }

}

static void InsertTargetPointRelativeToBasisIntoModelHashTable
  (Point tprb, ModelNum modelNum, Basis basis)
{

  HashTableEntryPtr ep =
    New_Set_HashTableEntry
      (
        _Dimension_,
        _BasisDimension_,
        tprb,
        modelNum,
        basis,
        &_ModelEntryPdf,
```

```
    NULL, /* NOT IMPLEMENTED YET */
    _FeatureTypePoint_
   );

 Insert_HashTablePtr_EntryPtr
 (
  _ModelHashTablePtr, ep,
  _InsertEntryPartitionRangeMinIndex, _InsertEntryPartitionRangeMaxIndex
 );

}

static void ComputeTargetPointRelativeToBasis
 (TargetPointsPtr tpp, Basis basis, Point tp, Point tprb)
{

 FeatureNum bfn0 = basis[0];
 FeatureNum bfn1 = basis[1];
 FeatureNum bfn2 = basis[2];

 Vector p0 = tpp->points[bfn0];
 Vector p1 = tpp->points[bfn1];
 Vector p2 = tpp->points[bfn2];

 Vector d1 = New_Vector(_Dimension_);
 Vector d2 = New_Vector(_Dimension_);
 Vector d3 = New_Vector(_Dimension_);
 Vector d4 = New_Vector(_Dimension_);

 Element n1;
 Element n3;
 Element n4;

 Vector v0 = New_Vector(_Dimension_);
 Vector v1 = New_Vector(_Dimension_);
 Vector v2 = New_Vector(_Dimension_);

 Vector d = New_Vector(_Dimension_);

 Sub_Vector_Vector(_Dimension_, p1, p0, d1);
 n1 = Get_2Norm_Vector(_Dimension_, d1);
 Mult_Vector_Scalar(_Dimension_, d1, 1.0/n1, v0);

 Sub_Vector_Vector(_Dimension_, p2, p0, d2);
 CrossProduct_Vector_Vector_3(d1, d2, d3);
 n3 = Get_2Norm_Vector(_Dimension_, d3);
 Mult_Vector_Scalar(_Dimension_, d3, 1.0/n3, v1);

 CrossProduct_Vector_Vector_3(d1, d3, d4);
 n4 = Get_2Norm_Vector(_Dimension_, d4);
 Mult_Vector_Scalar(_Dimension_, d4, 1.0/n4, v2);
```

```
 Sub_Vector_Vector(_Dimension_, tp, p0, d);

 tprb[0] = DotProduct_Vector_Vector(_Dimension_, d, v0);
 tprb[1] = DotProduct_Vector_Vector(_Dimension_, d, v1);
 tprb[2] = DotProduct_Vector_Vector(_Dimension_, d, v2);

 Destroy_Vector(d1);
 Destroy_Vector(d2);
 Destroy_Vector(d3);
 Destroy_Vector(d4);

 Destroy_Vector(v0);
 Destroy_Vector(v1);
 Destroy_Vector(v2);

 Destroy_Vector(d);

}

static void ComputeTargetPointsRelativeToBasis
 (TargetPointsPtr tpp, Basis basis, TargetPointsPtr tprbp)
{

 int i;

 for (i = 0; i < tpp->numPoints; i++) {
  ComputeTargetPointRelativeToBasis
   (tpp, basis, tpp->points[i], tprbp->points[i]);
 }

 tprbp->numPoints = tpp->numPoints;

}

static void InsertTargetPointsRelativeToBasisIntoModelHashTable
 (TargetPointsPtr tprbp, ModelNum modelNum, Basis basis)
{

 int i;
 FeatureNum basisFeatureNum0 = basis[0];
 FeatureNum basisFeatureNum1 = basis[1];
 FeatureNum basisFeatureNum2 = basis[2];

 for (i = 0; i < tprbp->numPoints; i++) {
  Point tprb = tprbp->points[i];

  InsertTargetPointRelativeToBasisIntoModelHashTable
   (tprb, modelNum, basis);

if (4 <= _MsgVerbosity_) {
```

```
  printf("Inserted target point relative to basis: point=(%d,%d,%d), modelNum=%d,
basis=(%d,%d,%d)\n",
    (int)tprb[0],
    (int)tprb[1],
    (int)tprb[2],
    modelNum,
    basisFeatureNum0,
    basisFeatureNum1,
    basisFeatureNum2
    );
  fflush(stdout);
}

  }

}

static void ComputeAndInsertTargetPointsRelativeToBasisIntoModelHashTable
  (TargetPointsPtr tpp, TargetPointsPtr tprbp, ModelNum modelNum)
{

  int tppNumPoints = tpp->numPoints;

  int i;
  int j;
  int k;

  int iStart = 0;
  int iEnd = (int)((float)tppNumPoints * (2.0/3.0));

  int jStart = iEnd + 1;
  int jEnd = (tppNumPoints - 1) - 1;

  for (i = iStart; i <= iEnd; i++)
  for (j = jStart; j <= jEnd; j++)
  for (k = (j+1); k <= (tppNumPoints - 1); k++)
  {
    FeatureNum basisFeatureNum0 = i;
    FeatureNum basisFeatureNum1 = j;
    FeatureNum basisFeatureNum2 = k;
    Basis basis = New_Basis(_BasisDimension_);
    basis[0] = basisFeatureNum0;
    basis[1] = basisFeatureNum1;
    basis[2] = basisFeatureNum2;
    ComputeTargetPointsRelativeToBasis(tpp, basis, tprbp);
    InsertTargetPointsRelativeToBasisIntoModelHashTable
      (tprbp, modelNum, basis);
    Destroy_Basis(basis);
  }

}
```

```
static void InsertModelIntoModelHashTable
 (
  TargetPointsPtr tpp, TargetPointsPtr tprbp,
  CharPtr modelType, CharPtr targetPoseX, CharPtr targetsFileName
 )
{

 int i;

 for (i = 0; i < 360; i += _ModelDegreeIncrement_) {
  String targetPoseZ;
  sprintf(targetPoseZ, "%d", i);
  strcpy(ModelTypesPoses[_ModelNum].modelType, modelType);
  strcpy(ModelTypesPoses[_ModelNum].poseX, targetPoseX);
  strcpy(ModelTypesPoses[_ModelNum].poseZ, targetPoseZ);
  ReadTargetPointsFromTargetsFile
   (tpp, modelType, targetPoseX, targetPoseZ, targetsFileName, TRUE);
  ComputeAndInsertTargetPointsRelativeToBasisIntoModelHashTable
   (tpp, tprbp, _ModelNum);
  _ModelNum++;
if (0 <= _MsgVerbosity_) {
 printf("Inserted model=(type=%s, poseX=%s, poseZ=%s) from %s\n",
  modelType,
  targetPoseX,
  targetPoseZ,
  targetsFileName
  );
 fflush(stdout);
}
 }

}
static void CreateModelHashTable()
{

 TargetPoints targetPoints;
 TargetPoints targetPointsRelativeToBasis;

PrintMsg(0, "Creating and initializing empty ModelHashTable ...\n");
 _ModelHashTablePtr =
  New_HashTable
   (
    _Dimension_,
    _DimensionMinVals,
    _DimensionMaxVals,
    _DimensionNumPartitions
   );
PrintMsg(0, "Created and initialized empty ModelHashTable\n");
if (0 <= _MsgVerbosity_) {
```

```
  printf("ModelHashTable: dimension=%d; minVals=%d,%d,%d; maxVals=%d,%d,%d,
numPartitions=%d,%d,%d; partitionSizes=%.2f,%.2f,%.2f\n",
    _Dimension_,
    (int)_DimensionMinVals[0],
    (int)_DimensionMinVals[1],
    (int)_DimensionMinVals[2],
    (int)_DimensionMaxVals[0],
    (int)_DimensionMaxVals[1],
    (int)_DimensionMaxVals[2],
    (int)_DimensionNumPartitions[0],
    (int)_DimensionNumPartitions[1],
    (int)_DimensionNumPartitions[2],
    Get_HashTablePtr_DimensionPartitionSizeI(_ModelHashTablePtr, 0),
    Get_HashTablePtr_DimensionPartitionSizeI(_ModelHashTablePtr, 1),
    Get_HashTablePtr_DimensionPartitionSizeI(_ModelHashTablePtr, 2)
    );
  fflush(stdout);
}

  Init_TargetPoints(&targetPoints);
  Init_TargetPoints(&targetPointsRelativeToBasis);

  _ModelNum = 0;


  InsertModelIntoModelHashTable(&targetPoints, &targetPointsRelativeToBasis,
    _ModelTypes[0], "0", _TargetsFileName);

  InsertModelIntoModelHashTable(&targetPoints, &targetPointsRelativeToBasis,
    _ModelTypes[1], "0", _TargetsFileName);

  InsertModelIntoModelHashTable(&targetPoints, &targetPointsRelativeToBasis,
    _ModelTypes[2], "0", _TargetsFileName);

  InsertModelIntoModelHashTable(&targetPoints, &targetPointsRelativeToBasis,
    _ModelTypes[3], "0", _TargetsFileName);

  UnInit_TargetPoints(&targetPoints);
  UnInit_TargetPoints(&targetPointsRelativeToBasis);

}

/*
static void ComputeVotesInModelHashTableForEntry
  (HashTableEntryPtr ep)
{

    HashTableEntryListPtr entryLp;

    entryLp =
      GetHashTableEntryListForEntry(_ModelHashTablePtr, ep);
```

```
  ComputeVotesInHashTableEntryListForEntry(_ModelHashTablePtr, entryLp, ep);

}
*/

static void ComputeVotesInModelHashTableForEntry
 (HashTableEntryPtr ep)
{

 ComputeVotesInHashTableForEntry
  (
   _ModelHashTablePtr, ep,
   _ComputeVotePartitionRangeMinIndex, _ComputeVotePartitionRangeMaxIndex
  );

}

static void ComputeVotesInModelHashTableForTargetPointRelativeToBasis
 (Point tprb, Basis basis)
{

 HashTableEntryPtr ep =
  New_Set_HashTableEntry
   (
    _Dimension_,
    _BasisDimension_,
    tprb,
    _ModelNumNull_,
    basis,
    NULL,
    NULL,
    _FeatureTypePoint_
   );

 ComputeVotesInModelHashTableForEntry(ep);

}

static void ComputeVotesInModelHashTableForTargetPointsRelativeToBasis
 (TargetPointsPtr tprbp, Basis basis)
{

 int i;

 for (i = 0; i < tprbp->numPoints; i++) {

  Point tprb = tprbp->points[i];

  ComputeVotesInModelHashTableForTargetPointRelativeToBasis
   (tprb, basis);
```

```
 }

}

static void HistogramVotesInModelHashTable()
{

 int numBuckets = Get_HashTablePtr_NumBuckets(_ModelHashTablePtr);
 int i;
 int j0;
 int j1;
 int j2;

PrintMsg(0, "Histogramming votes in model hash table ...\n");

 /* Initialize Votes. */
 for (i = 0; i < _NumModels_; i++) {
  for (j0 = 0; j0 < _MaxNumBasesDim0_; j0++) {
  for (j1 = 0; j1 < _MaxNumBasesDim1_; j1++) {
  for (j2 = 0; j2 < _MaxNumBasesDim2_; j2++) {
   Votes[i][j0][j1][j2] = _VoteNull_;
  }
  }
  }
 }

 /* Initialize NumbersOfModelEntriesThatContributedToVote. */
 for (i = 0; i < _NumModels_; i++) {
  for (j0 = 0; j0 < _MaxNumBasesDim0_; j0++) {
  for (j1 = 0; j1 < _MaxNumBasesDim1_; j1++) {
  for (j2 = 0; j2 < _MaxNumBasesDim2_; j2++) {
   NumbersOfModelEntriesThatContributedToVote[i][j0][j1][j2] = 0;
  }
  }
  }
 }

 /*
   Histogram votes for matching model entries in model hash table.
 */
 for (i = 0; i < numBuckets; i++) {
  HashTableBucketPtr bp = Get_HashTablePtr_BucketPtrI(_ModelHashTablePtr, i);
  HashTableEntryListPtr lp = bp;
  if (Get_HashTableEntryListPtr_VotesHaveBeenComputed(lp)) {
   HashTableEntryNodePtr lnp = Get_HashTableEntryListPtr_FirstNodePtr(lp);
   while (lnp != NULL) {

    Vote lnpVote = Get_HashTableEntryNodePtr_Vote(lnp);
    HashTableEntryPtr lep = Get_HashTableEntryNodePtr_EntryPtr(lnp);
    ModelNum lepModelNum = Get_HashTableEntryPtr_ModelNum(lep);
```

```
      Basis lepBasis = Get_HashTableEntryPtr_Basis(lep);
      FeatureNum lepBasisFeatureNum0 = lepBasis[0];
      FeatureNum lepBasisFeatureNum1 = lepBasis[1];
      FeatureNum lepBasisFeatureNum2 = lepBasis[2];

      if (lnpVote == _VoteNull_) {
       /* model entry didn't match */

       /* do nothing */
      } else {
       /* model entry matched */

       if (
           Votes
            [lepModelNum]
            [lepBasisFeatureNum0]
            [lepBasisFeatureNum1]
            [lepBasisFeatureNum2]
           == _VoteNull_
         ) {
         Votes
          [lepModelNum]
          [lepBasisFeatureNum0]
          [lepBasisFeatureNum1]
          [lepBasisFeatureNum2]
         = lnpVote;
       } else {
         Votes
          [lepModelNum]
          [lepBasisFeatureNum0]
          [lepBasisFeatureNum1]
          [lepBasisFeatureNum2]
         += lnpVote;
       }
       NumbersOfModelEntriesThatContributedToVote
        [lepModelNum]
        [lepBasisFeatureNum0]
        [lepBasisFeatureNum1]
        [lepBasisFeatureNum2]
       ++;
      }

      lnp = Get_HashTableEntryNodePtr_NextPtr(lnp);

    }
   }
  }

  /*
    Add penalties to histogram for unmatched model entries in model hash table.
  */
```

```
 for (i = 0; i < _NumModels_; i++) {
  for (j0 = 0; j0 < _MaxNumBasesDim0_; j0++) {
  for (j1 = 0; j1 < _MaxNumBasesDim1_; j1++) {
  for (j2 = 0; j2 < _MaxNumBasesDim2_; j2++) {
    if (Votes[i][j0][j1][j2] == _VoteNull_) {
     /* do nothing */
    } else {
     int numUnmatchedModelPoints =
      (
       NumbersOfModelPointsInModel[i]
        -
       NumbersOfModelEntriesThatContributedToVote[i][j0][j1][j2]
      );
     Votes[i][j0][j1][j2] -=
      numUnmatchedModelPoints
       * _VotePenaltyForUnmatchedModelPoint_;
    }
   }
   }
   }
 }

PrintMsg(0, "Histogrammed votes in model hash table\n");

}

static void ComputeAndPrintWinningModelBasisPairsInHistogram()
{

 int i;
 int j0;
 int j1;
 int j2;
 int j;
 int k;

 printf("Computing top %d model, basis pairs sorted on vote ...\n",
   _NumWinningModelBasisPairs_);

 /* Initialize ModelNumBasisVoteArray. */
 k = 0;
 for (i = 0; i < _NumModels_; i++) {
  for (j0 = 0; j0 < _MaxNumBasesDim0_; j0++) {
  for (j1 = 0; j1 < _MaxNumBasesDim1_; j1++) {
  for (j2 = 0; j2 < _MaxNumBasesDim2_; j2++) {
    ModelNumBasisVoteArray[k].modelNum = i;
    ModelNumBasisVoteArray[k].basisFeatureNum0 = j0;
    ModelNumBasisVoteArray[k].basisFeatureNum1 = j1;
    ModelNumBasisVoteArray[k].basisFeatureNum2 = j2;
    ModelNumBasisVoteArray[k].vote = Votes[i][j0][j1][j2];
    k++;
```

```
    }
    }
    }
  }

  /*
    Sort ModelNumBasisVoteArray, but to get only the top
     _NumWinningModelBasisPairs_.
  */
  for (i = 0; i <= (_NumWinningModelBasisPairs_-1); i++) {

    // Find min in rest of array.
    int minIndex = i;
    for (j = i; j <= (ModelNumBasisVoteArraySize-1); j++) {
      if
        (
          ModelNumBasisVoteArray[j].vote
            > ModelNumBasisVoteArray[minIndex].vote
        ) {
        minIndex = j;
      }
    }

    // Swap current and min.
    {
    ModelNumBasisVote temp = ModelNumBasisVoteArray[i];
    ModelNumBasisVoteArray[i] = ModelNumBasisVoteArray[minIndex];
    ModelNumBasisVoteArray[minIndex] = temp;
    }

  }

  /*
    Print top _NumWinningModelBasisPairs_ elements of ModelNumBasisVoteArray.
  */
  if (0 <= _MsgVerbosity_) {
    printf("Top %d model, basis pairs sorted on vote:\n",
      _NumWinningModelBasisPairs_);
    for (i = 0; i < _NumWinningModelBasisPairs_; i++) {
      if (ModelNumBasisVoteArray[i].vote != _VoteNull_) {
        printf("modelNum=%2d,basis=(%2d,%2d,%2d),vote=%8.2f, %2d/%2d pts matched,
%s,%4s,%4s\n",
          ModelNumBasisVoteArray[i].modelNum,
          ModelNumBasisVoteArray[i].basisFeatureNum0,
          ModelNumBasisVoteArray[i].basisFeatureNum1,
          ModelNumBasisVoteArray[i].basisFeatureNum2,
          ModelNumBasisVoteArray[i].vote,
          NumbersOfModelEntriesThatContributedToVote
            [ModelNumBasisVoteArray[i].modelNum]
            [ModelNumBasisVoteArray[i].basisFeatureNum0]
            [ModelNumBasisVoteArray[i].basisFeatureNum1]
```

```
        [ModelNumBasisVoteArray[i].basisFeatureNum2],
       NumbersOfModelPointsInModel
        [ModelNumBasisVoteArray[i].modelNum],
       ModelTypesPoses[ModelNumBasisVoteArray[i].modelNum].modelType,
       ModelTypesPoses[ModelNumBasisVoteArray[i].modelNum].poseX,
       ModelTypesPoses[ModelNumBasisVoteArray[i].modelNum].poseZ
       );
    }
  }
 }

}

/*
 main()
*/

main(int argc, char *argv[])
{

 TargetPoints testTargetPoints;
 float testTargetPlusMinusRandomUniformNoise;
 TargetPoints testTargetPointsRelativeToBasis;
 FeatureNum testTargetBasisFeatureNum0;
 FeatureNum testTargetBasisFeatureNum1;
 FeatureNum testTargetBasisFeatureNum2;
 Basis testTargetBasis;

 int modelEntryPdfType;
 int modelEntryPdfDimension;
 Point modelEntryPdfMean;
 Matrix modelEntryPdfCov;
 Element modelEntryPdfDetCov;
 Matrix modelEntryPdfInvCov;
 Element modelEntryPdfCovSigma0;
 Element modelEntryPdfCovSigma1;
 Element modelEntryPdfCovSigma2;
PrintMsg(0, "Entered main()\n");

 /* Check and process arguments. */

 if ( (argc == 8) || (argc == 9) ) {

  strcpy(_TargetsFileName, argv[1]);
  strcpy(_TestTargetType, argv[2]);
  strcpy(_TestTargetPoseX, argv[3]);
  strcpy(_TestTargetPoseZ, argv[4]);
  strcpy(_TestTargetBasisFeatureNum0, argv[5]);
  strcpy(_TestTargetBasisFeatureNum1, argv[6]);
  strcpy(_TestTargetBasisFeatureNum2, argv[7]);
  if (argc == 9) {
```

```
    strcpy(_TestTargetPlusMinusRandomUniformNoise, argv[8]);
   } else {
    strcpy(_TestTargetPlusMinusRandomUniformNoise, "0");
   }

  } else {

   printf("%s: Incorrect arguments\n", argv[0]);
   printf("Usage: %s <targetsFileName> <testTargetType> <testTargetPoseX> <testTargetPoseZ>
<testTargetBasisFeatureNum0> <testTargetBasisFeatureNum1> <testTargetBasisFeatureNum2>
<testTargetPlusMinusRandomUniformNoise><optional><default=0>\n",
    argv[0]);
   exit(0);

  }


  /*
   Initialize _ModelEntryPdf.
  */

  modelEntryPdfType = _PDFTypeGaussian_;
  Set_PDFPtr_Type(&_ModelEntryPdf, modelEntryPdfType);

  modelEntryPdfDimension = _Dimension_;
  Set_PDFPtr_Dimension(&_ModelEntryPdf, modelEntryPdfDimension);

  modelEntryPdfMean = New_Point(_Dimension_);
  modelEntryPdfMean[0] = 0;
  modelEntryPdfMean[1] = 0;
  modelEntryPdfMean[2] = 0;
  Set_PDFPtr_Mean(&_ModelEntryPdf, modelEntryPdfMean);

  modelEntryPdfCovSigma0 = 5.0;
  modelEntryPdfCovSigma1 = 5.0;
  modelEntryPdfCovSigma2 = 5.0;

  modelEntryPdfCov = New_Matrix(_Dimension_, _Dimension_);
  Set_Matrix_ElementIJ
   (modelEntryPdfCov, _Dimension_, _Dimension_, 0, 0,
    (modelEntryPdfCovSigma0*modelEntryPdfCovSigma0));
  Set_Matrix_ElementIJ
   (modelEntryPdfCov, _Dimension_, _Dimension_, 0, 1, 0);
  Set_Matrix_ElementIJ
   (modelEntryPdfCov, _Dimension_, _Dimension_, 0, 2, 0);
  Set_Matrix_ElementIJ
   (modelEntryPdfCov, _Dimension_, _Dimension_, 1, 0, 0);
  Set_Matrix_ElementIJ
   (modelEntryPdfCov, _Dimension_, _Dimension_, 1, 1,
    (modelEntryPdfCovSigma1*modelEntryPdfCovSigma1));
  Set_Matrix_ElementIJ
```

```
 (modelEntryPdfCov, _Dimension_, _Dimension_, 1, 2, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfCov, _Dimension_, _Dimension_, 2, 0, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfCov, _Dimension_, _Dimension_, 2, 1, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfCov, _Dimension_, _Dimension_, 2, 2,
  (modelEntryPdfCovSigma2*modelEntryPdfCovSigma2));
Set_PDFPtr_Cov(&_ModelEntryPdf, modelEntryPdfCov);

modelEntryPdfDetCov = 25*25*25;
Set_PDFPtr_DetCov(&_ModelEntryPdf, modelEntryPdfDetCov);

modelEntryPdfInvCov = New_Matrix(_Dimension_, _Dimension_);
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 0, 0,
  1.0/(modelEntryPdfCovSigma0*modelEntryPdfCovSigma0));
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 0, 1, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 0, 2, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 1, 0, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 1, 1,
  1.0/(modelEntryPdfCovSigma1*modelEntryPdfCovSigma1));
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 1, 2, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 2, 0, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 2, 1, 0);
Set_Matrix_ElementIJ
 (modelEntryPdfInvCov, _Dimension_, _Dimension_, 2, 2,
  1.0/(modelEntryPdfCovSigma2*modelEntryPdfCovSigma2));
Set_PDFPtr_InvCov(&_ModelEntryPdf, modelEntryPdfInvCov);

/* Create model hash table. */

CreateModelHashTable();

Init_TargetPoints(&testTargetPoints);
Init_TargetPoints(&testTargetPointsRelativeToBasis);


/* Read test target from test targets file. */

ReadTargetPointsFromTargetsFile
 (
  &testTargetPoints,
  _TestTargetType, _TestTargetPoseX, _TestTargetPoseZ,
```

```
    _TargetsFileName,
    FALSE
   );

if (0 <= _MsgVerbosity_) {
  printf("Read test target = (type=%s, poseX=%s, poseZ=%s) from %s:\n",
    _TestTargetType, _TestTargetPoseX, _TestTargetPoseZ,
    _TargetsFileName
   );
  fflush(stdout);
}
Print_TargetPoints(0, &testTargetPoints);

  /* Add plus or minus random uniform noise to test target. */

  sscanf(_TestTargetPlusMinusRandomUniformNoise, "%f",
    &testTargetPlusMinusRandomUniformNoise);

  AddPlusMinusRandomUniformNoiseToTargetPoints
    (&testTargetPoints, testTargetPlusMinusRandomUniformNoise);


if (0 <= _MsgVerbosity_) {
  printf("Added -%.2f..%.2f random uniform noise to test target = (type=%s, poseX=%s, poseZ=%s) from
%s:\n",
    testTargetPlusMinusRandomUniformNoise,
    testTargetPlusMinusRandomUniformNoise,
    _TestTargetType, _TestTargetPoseX, _TestTargetPoseZ,
    _TargetsFileName
   );
  fflush(stdout);
}
Print_TargetPoints(0, &testTargetPoints);

  /* Choose basis for test target. */

  sscanf(_TestTargetBasisFeatureNum0, "%d", &testTargetBasisFeatureNum0);
  sscanf(_TestTargetBasisFeatureNum1, "%d", &testTargetBasisFeatureNum1);
  sscanf(_TestTargetBasisFeatureNum2, "%d", &testTargetBasisFeatureNum2);
  testTargetBasis = New_Basis(_BasisDimension_);
  testTargetBasis[0] = testTargetBasisFeatureNum0;
  testTargetBasis[1] = testTargetBasisFeatureNum1;
  testTargetBasis[2] = testTargetBasisFeatureNum2;

if (0 <= _MsgVerbosity_) {
  printf("Chose basis=(%d,%d,%d)\n",
    testTargetBasis[0],
    testTargetBasis[1],
    testTargetBasis[2]
   );
  fflush(stdout);
```

```
}

 /* Compute test target relative to chosen basis. */

 ComputeTargetPointsRelativeToBasis
  (
   &testTargetPoints, testTargetBasis, &testTargetPointsRelativeToBasis
  );

if (0 <= _MsgVerbosity_) {
 printf("Computed test target relative to chosen basis:\n");
 fflush(stdout);
}
Print_TargetPoints(0, &testTargetPointsRelativeToBasis);

 /*
  Compute votes in model hash table for test target relative to chosen basis.
 */

 ComputeVotesInModelHashTableForTargetPointsRelativeToBasis
  (&testTargetPointsRelativeToBasis, testTargetBasis);

 Destroy_Basis(testTargetBasis);

 UnInit_TargetPoints(&testTargetPoints);
 UnInit_TargetPoints(&testTargetPointsRelativeToBasis);

 /* Histogram votes in model hash table. */

 HistogramVotesInModelHashTable();

 /* Compute and print winning model, basis pairs in histogram. */

 ComputeAndPrintWinningModelBasisPairsInHistogram();

PrintMsg(0, "Exiting main()\n");

}
```

```
/*
 matrix.c
*/

#include "matrix.h"

extern Vector New_Vector(int dimension)
{
 Vector vector = (Vector)malloc(dimension*sizeof(Element));
 return vector;
```

```
}

extern void Destroy_Vector(Vector vector)
{
 free(vector);
}
extern void Assign_Vector_Vector(int dimension, Vector a, Vector b)
{
 int i;
 for (i = 0; i < dimension; i++) {
  b[i] = a[i];
 }
}

extern void Add_Vector_Vector(int dimension, Vector a, Vector b, Vector c)
{
 int i;
 for (i = 0; i < dimension; i++) {
  c[i] = a[i] + b[i];
 }
}

extern void Sub_Vector_Vector(int dimension, Vector a, Vector b, Vector c)
{
 int i;
 for (i = 0; i < dimension; i++) {
  c[i] = a[i] - b[i];
 }
}

extern Element Get_2Norm_Vector(int dimension, Vector a)
{
 Element norm2 = 0;
 int i;
 for (i = 0; i < dimension; i++) {
  norm2 += a[i]*a[i];
 }
 norm2 = sqrt(norm2);
 return norm2;
}

extern void Mult_Vector_Scalar(int dimension, Vector a, Element e, Vector b)
{
 int i;
 for (i = 0; i < dimension; i++) {
  b[i] = e * a[i];
 }
}

extern Element DotProduct_Vector_Vector
```

```
 (int dimension, Vector a, Vector b)
{
 Element dotProduct = 0;
 int i;
 for (i = 0; i < dimension; i++) {
  dotProduct += a[i] * b[i];
 }
 return dotProduct;
}

extern void CrossProduct_Vector_Vector_3
 (Vector a, Vector b, Vector c)
{
 c[0] = a[1]*b[2] - a[2]*b[1];
 c[1] = a[0]*b[2] - a[2]*b[0];
 c[2] = a[0]*b[1] - a[1]*b[0];
}

extern Matrix New_Matrix(int numRows, int numCols)
{
 int numElements = numRows*numCols;
 Matrix matrix = (Matrix)malloc(numElements*sizeof(Element));
 return matrix;
}

extern void Destroy_Matrix(Matrix matrix)
{
 free(matrix);
}

extern void Copy_Matrix(Matrix a, Matrix b, int numRows, int numCols)
{
 int numElements = numRows*numCols;
 int i;

 for (i = 0; i < numElements; i++) {
  b[i] = a[i];
 }
}

/*
extern float det_matrix_3x3(matrix_3x3 m)
{
 return m[0][0] * m[1][1] * m[2][2] +
      m[0][1] * m[1][2] * m[2][0] +
      m[0][2] * m[1][0] * m[2][1] -
      m[0][2] * m[1][1] * m[2][0] -
      m[0][0] * m[1][2] * m[2][1] -
      m[0][1] * m[1][0] * m[2][2];
}
```

```
extern float det_matrix_4x4(matrix_4x4 m)
{
 matrix_3x3 m1;
 matrix_3x3 m2;
 matrix_3x3 m3;
 matrix_3x3 m4;

 int i;
 int j;

 for (i = 0 ; i < 3 ; i++)
  for (j = 0 ; j < 3 ; j++) {
   m1[i][j] = m[i + 1][j + 1];
   m2[i][j] = m[i + 1][j + (j > 0)];
   m3[i][j] = m[i + 1][j + (j > 1)];
   m4[i][j] = m[i + 1][j];
  }

 return m[0][0] * det_matrix_3x3(m1)
       - m[0][1] * det_matrix_3x3(m2)
       + m[0][2] * det_matrix_3x3(m3)
       - m[0][3] * det_matrix_3x3(m4);
}
*/

/*
extern int solve_4x4(vector_4 x, matrix_4x4 A, vector_4 z)
{

 float detA;

 matrix_4x4 M1;
 matrix_4x4 M2;
 matrix_4x4 M3;
 matrix_4x4 M4;

 float detM1;
 float detM2;
 float detM3;
 float detM4;

 detA = det_matrix_4x4(A);
 if (detA == 0) {
  return FALSE;
 }

 copy_matrix_4x4(A, M1);
 M1[0][0] = z[0];
 M1[1][0] = z[1];
 M1[2][0] = z[2];
 M1[3][0] = z[3];
```

```
 detM1 = det_matrix_4x4(M1);

 copy_matrix_4x4(A, M2);
 M2[0][1] = z[0];
 M2[1][1] = z[1];
 M2[2][1] = z[2];
 M2[3][1] = z[3];
 detM2 = det_matrix_4x4(M2);

 copy_matrix_4x4(A, M3);
 M3[0][2] = z[0];
 M3[1][2] = z[1];
 M3[2][2] = z[2];
 M3[3][2] = z[3];
 detM3 = det_matrix_4x4(M3);

 copy_matrix_4x4(A, M4);
 M4[0][3] = z[0];
 M4[1][3] = z[1];
 M4[2][3] = z[2];
 M4[3][3] = z[3];
 detM4 = det_matrix_4x4(M4);

 x[0] = detM1/detA;
 x[1] = detM2/detA;
 x[2] = detM3/detA;
 x[3] = detM4/detA;

 return TRUE;
}
*/

extern void Mult_Vector_Matrix
 (Vector v, int vDim, Matrix m, int mNumRows, int mNumCols, Vector w)
{
 int i;
 int j;

 Assert(vDim == mNumRows, "Mult_Vector_Matrix()");

 for (j = 0; j < mNumCols; j++) {
  Element wj = 0;

  for (i = 0; i < mNumRows; i++) {
   Element t1 = Get_Vector_ElementI(v, i);
   Element t2 = Get_Matrix_ElementIJ(m, mNumRows, mNumCols, i, j);
   Element product = t1*t2;
/*
printf("t1=%f, t2=%f, product=%f\n", t1, t2, product); fflush(stdout);
*/
   wj += product;
```

```
  }
  Set_Vector_ElementI(w, j, wj);
 }
}

extern void Mult_Matrix_Vector
 (Matrix m, int mNumRows, int mNumCols, Vector v, int vDim, Vector w)
{
 int i;
 int j;

 Assert(mNumCols == vDim, "Mult_Matrix_Vector()");

 for (i = 0; i < mNumRows; i++) {
  Element wi = 0;

  for (j = 0; j < mNumCols; j++) {
   wi += Get_Vector_ElementI(v, j)
        * Get_Matrix_ElementIJ(m, mNumRows, mNumCols, i, j);
  }
  Set_Vector_ElementI(w, i, wi);
 }
}

extern void Mult_Vector_Vector(Vector v1, Vector v2, int vDim, ElementPtr wp)
{
 Element w = 0;
 int j;
 for (j = 0; j < vDim; j++) {
  w += Get_Vector_ElementI(v1, j)
      * Get_Vector_ElementI(v2, j);
 }
 *wp = w;
}
```

```
#ifndef _hashtbl_h_
#define _hashtbl_h_

/*
 hashtbl.h

 This module implements the n-dimensional hash table data type,
 which includes hash table entries.

 This module also implements compute vote functions.

 Search for the comment "HashTable Creation and Access Functions"
 to locate the hash table creation and access functions.
```

```
 Search for the comment "Compute Vote Functions"
 to locate the compute vote functions.
*/

#include "utils.h"
#include "matrix.h"

/*
 DimensionType
*/

typedef float DimensionType;

typedef DimensionType *DimensionTypePtr;

/*
 Point
*/

typedef DimensionTypePtr Point; /* array dimension of DimensionType */

extern Point New_Point(int dimension);
extern void Destroy_Point(Point point);
extern void Copy_Point(int dimension, Point p1, Point p2); /* Copy p1 to p2. */
extern DimensionType Get_Distance_Point_Point
 (int dimension, Point p1, Point p2);

/* Compute p3 = p1 + p2. */
extern void Add_Point_Point(int dimension, Point p1, Point p2, Point p3);

/* Compute p3 = p1 - p2. */
extern void Subtract_Point_Point(int dimension, Point p1, Point p2, Point p3);

/*
 ModelNum
*/

typedef int ModelNum;

#define _ModelNumNull_ -1

/*
 FeatureNum
*/
typedef int FeatureNum;
typedef FeatureNum *FeatureNumPtr;

/*
 Basis
*/
```

```
typedef FeatureNumPtr Basis;  /* array dimension of FeatureNum */

extern Basis New_Basis(int dimension);

extern void Destroy_Basis(Basis basis);

extern void Copy_Basis(int dimension, Basis b1, Basis b2); /* Copy b1 to b2. */

/*
 PDFType
 PDF
*/

typedef int PDFType;

/* PDFType-s */
#define _PDFTypeGaussian_ 0
typedef struct PDF
  {
   PDFType type;
   int dimension;
   Point mean;
   Matrix cov;     /* covariance matrix */
   Element detCov; /* determinant of covariance matrix */
   Matrix invCov;  /* inverse of covariance matrix */
  } PDF;

#define Get_PDFPtr_Type(pp) \
  ((pp)->type)
#define Get_PDFPtr_Dimension(pp) \
  ((pp)->dimension)
#define Get_PDFPtr_Mean(pp) \
  ((pp)->mean)
#define Get_PDFPtr_Cov(pp) \
  ((pp)->cov)
#define Get_PDFPtr_DetCov(pp) \
  ((pp)->detCov)
#define Get_PDFPtr_InvCov(pp) \
  ((pp)->invCov)

#define Set_PDFPtr_Type(pp, t) \
  { (pp)->type = (t); }
#define Set_PDFPtr_Dimension(pp, d) \
  { (pp)->dimension = (d); }
#define Set_PDFPtr_Mean(pp, m) \
  { (pp)->mean = (m); }
#define Set_PDFPtr_Cov(pp, c) \
  { (pp)->cov = (c); }
#define Set_PDFPtr_DetCov(pp, dc) \
  { (pp)->detCov = (dc); }
#define Set_PDFPtr_InvCov(pp, ic) \
```

```
 { (pp)->invCov = (ic); }

typedef PDF *PDFPtr;

/*
 NOT IMPLEMENTED YET.
 Stats
*/

typedef int Stats;

typedef Stats *StatsPtr;

/*
 FeatureType
*/

typedef int FeatureType;

/* FeatureType-s */
#define _FeatureTypePoint_ 0

/*
 Vote
*/

/* A non-null vote is >= _VoteMin_. */
typedef float Vote;

#define _VoteMin_  0

#define _VoteNull_ -1

/*
 HashTableEntry
*/

typedef struct HashTableEntry
  {
   Point       point;
   ModelNum    modelNum;
   Basis       basis;
   PDFPtr      pdfPtr;
   StatsPtr    statsPtr;
   FeatureType featureType;
   Vote vote;
  } HashTableEntry;

#define Get_HashTableEntryPtr_Point(ep) \
  ((ep)->point)
```

```
#define Get_HashTableEntryPtr_ModelNum(ep) \
 ((ep)->modelNum)

#define Get_HashTableEntryPtr_Basis(ep) \
 ((ep)->basis)

#define Get_HashTableEntryPtr_PdfPtr(ep) \
 ((ep)->pdfPtr)

#define Get_HashTableEntryPtr_StatsPtr(ep) \
 ((ep)->statsPtr)

#define Get_HashTableEntryPtr_FeatureType(ep) \
 ((ep)->featureType)

#define Get_HashTableEntryPtr_Vote(ep) \
 ((ep)->vote)

#define Set_HashTableEntryPtr_Point(ep, p) \
 { (ep)->point = (p); }

#define Set_HashTableEntryPtr_ModelNum(ep, mn) \
 { (ep)->modelNum = (mn); }

#define Set_HashTableEntryPtr_Basis(ep, b) \
 { (ep)->basis = (b); }

#define Set_HashTableEntryPtr_PdfPtr(ep, pp) \
 { (ep)->pdfPtr = (pp); }

#define Set_HashTableEntryPtr_StatsPtr(ep, sp) \
 { (ep)->statsPtr = (sp); }

#define Set_HashTableEntryPtr_FeatureType(ep, ft) \
 { (ep)->featureType = (ft); }

#define Set_HashTableEntryPtr_Vote(ep, v) \
 { (ep)->vote = (v); }

typedef HashTableEntry *HashTableEntryPtr;

extern HashTableEntryPtr New_HashTableEntry();

extern HashTableEntryPtr New_Set_HashTableEntry
 (
  int dimension,
  int basisDimension,
  Point point,
  ModelNum modelNum,
  Basis basis,
  PDFPtr pdfPtr,
```

```
    StatsPtr statsPtr,
    FeatureType featureType
  );

/*
  HashTableEntryNode
*/

typedef struct HashTableEntryNode *HashTableEntryNodePtr;

typedef struct HashTableEntryNode
  {
    HashTableEntryPtr entryPtr;
    DimensionType distance;  /* from this entry to center of this bucket */
/*
    Vote vote;
*/
    HashTableEntryNodePtr nextPtr;
  } HashTableEntryNode;

#define Get_HashTableEntryNodePtr_EntryPtr(np) \
  ((np)->entryPtr)

#define Get_HashTableEntryNodePtr_Distance(np) \
  ((np)->distance)

#define Get_HashTableEntryNodePtr_Vote(np) \
  ( Get_HashTableEntryPtr_Vote((np)->entryPtr) )
/*
  ((np)->vote)
*/

#define Get_HashTableEntryNodePtr_NextPtr(np) \
  ((np)->nextPtr)

#define Set_HashTableEntryNodePtr_EntryPtr(np, ep) \
  { (np)->entryPtr = (ep); }

#define Set_HashTableEntryNodePtr_Distance(np, d) \
  { (np)->distance = (d); }

#define Set_HashTableEntryNodePtr_Vote(np, v) \
  { Set_HashTableEntryPtr_Vote((np)->entryPtr, (v)); }
/*
  { (np)->vote = (v); }
*/

#define Set_HashTableEntryNodePtr_NextPtr(np, nextp) \
  {(np)->nextPtr = (nextp);}

extern HashTableEntryNodePtr New_HashTableEntryNode();
```

```
/*
 HashTableEntryList
*/

typedef struct HashTableEntryList
  {
   HashTableEntryNodePtr firstNodePtr;
   HashTableEntryNodePtr lastNodePtr;
   boolean votesHaveBeenComputed;
  } HashTableEntryList;

typedef HashTableEntryList *HashTableEntryListPtr;

#define Get_HashTableEntryListPtr_FirstNodePtr(lp) \
  ((lp)->firstNodePtr)

#define Get_HashTableEntryListPtr_LastNodePtr(lp) \
  ((lp)->lastNodePtr)

#define Get_HashTableEntryListPtr_VotesHaveBeenComputed(lp) \
  ((lp)->votesHaveBeenComputed)

#define Set_HashTableEntryListPtr_FirstNodePtr(lp, np) \
  { (lp)->firstNodePtr = (np); }

#define Set_HashTableEntryListPtr_LastNodePtr(lp, np) \
  { (lp)->lastNodePtr = (np); }

#define Set_HashTableEntryListPtr_VotesHaveBeenComputed(lp, b) \
  { (lp)->votesHaveBeenComputed = (b); }

extern HashTableEntryListPtr New_HashTableEntryList();

/*
  Insert entry *ep at the head position of entry list *lp.
  distance is the distance from entry *ep to the center of the bucket
  that entry list *lp represents.
*/
extern HashTableEntryListPtr InsertHead_HashTableEntryListPtr_EntryPtr_Distance
  (HashTableEntryListPtr lp, HashTableEntryPtr ep, DimensionType distance);

/*
  HashTableBucket
*/

typedef HashTableEntryList HashTableBucket;

typedef HashTableBucket *HashTableBucketPtr;

/*
```

```
 HashTable
*/

typedef struct HashTable
 {
  int dimension;
  DimensionType *dimensionMinVals;
  DimensionType *dimensionMaxVals;
  int *dimensionNumPartitions;
  DimensionType *dimensionPartitionSizes;
  int numBuckets;
  HashTableBucketPtr *bucketPtrs;
 } HashTable;

typedef HashTable *HashTablePtr;

#define Get_HashTablePtr_Dimension(htp) \
 ((htp)->dimension)

#define Get_HashTablePtr_DimensionMinValI(htp, i) \
 ((htp)->dimensionMinVals[(i)])

#define Get_HashTablePtr_DimensionMaxValI(htp, i) \
 ((htp)->dimensionMaxVals[(i)])

#define Get_HashTablePtr_DimensionNumPartitionI(htp, i) \
 ((htp)->dimensionNumPartitions[(i)])

#define Get_HashTablePtr_DimensionPartitionSizeI(htp, i) \
 ((htp)->dimensionPartitionSizes[(i)])

#define Get_HashTablePtr_NumBuckets(htp) \
 ((htp)->numBuckets)

#define Get_HashTablePtr_BucketPtrI(htp, i) \
 ((htp)->bucketPtrs[(i)])

#define Set_HashTablePtr_Dimension(htp, d) \
 ((htp)->dimension = (d))

#define Set_HashTablePtr_DimensionMinVals(htp, vs) \
 { (htp)->dimensionMinVals = (vs); }

#define Set_HashTablePtr_DimensionMaxVals(htp, vs) \
 { (htp)->dimensionMaxVals = (vs); }

#define Set_HashTablePtr_DimensionMinValI(htp, i, v) \
 { (htp)->dimensionMinVals[(i)] = (v); }

#define Set_HashTablePtr_DimensionMaxValI(htp, i, v) \
 { (htp)->dimensionMaxVals[(i)] = (v); }
```

```
#define Set_HashTablePtr_DimensionNumPartitions(htp, ps) \
  { (htp)->dimensionNumPartitions = (ps); }

#define Set_HashTablePtr_DimensionPartitionSizes(htp, pss) \
  { (htp)->dimensionPartitionSizes = (pss); }

#define Set_HashTablePtr_DimensionNumPartitionI(htp, i, p) \
  { (htp)->dimensionNumPartitions[(i)] = (p); }

#define Set_HashTablePtr_DimensionPartitionSizeI(htp, i, ps) \
  { (htp)->dimensionPartitionSizes[(i)] = (ps); }

#define Set_HashTablePtr_NumBuckets(htp, nb) \
  { (htp)->numBuckets = (nb); }

#define Set_HashTablePtr_BucketPtrs(htp, bps) \
  { (htp)->bucketPtrs = (bps); }

#define Set_HashTablePtr_BucketPtrI(htp, i, bp) \
  { (htp)->bucketPtrs[(i)] = (bp); }

/*
  HashTable Creation and Access Functions
*/

extern HashTablePtr New_HashTable
  (
    int dimension,
    DimensionType *dimensionMinVals,
    DimensionType *dimensionMaxVals,
    int *dimensionNumPartitions
  );

/*
  Determine the bucket of hash table *htp in which point "point" lands and
  compute the bucket's bucket number and bucket midpoint.
  Return the bucket number and set "bucketMidpoint" to the bucket midpoint.
*/
extern int Get_HashTablePtr_Point_BucketNum_BucketMidpoint
  (HashTablePtr htp, Point point, Point bucketMidpoint);

extern void InsertIntoHypercube_HashTablePtr_EntryPtr
  (
    HashTablePtr htp, HashTableEntryPtr ep,
    int partitionRangeMinIndex, int partitionRangeMaxIndex,
    Point entryPoint, int dimension, Point point, Point bucketMidpoint,
    int loopDimensionNum
  );

/*
```

```
 Insert entry *ep into hash table *htp.
 Insert the entry into all buckets in an n-dim. rectangular neighborhood
 of the entry, where the n-dim. rectangular neighborhood is defined by
 partitionRangeMinIndex and partitionRangeMaxIndex.
*/
extern HashTablePtr Insert_HashTablePtr_EntryPtr
 (
  HashTablePtr htp, HashTableEntryPtr ep,
  int partitionRangeMinIndex, int partitionRangeMaxIndex
 );

extern HashTableEntryListPtr GetHashTableEntryListForEntry
 (HashTablePtr htp, HashTableEntryPtr ep);

/*
 Compute Vote Functions
*/

extern void ComputeVotesInHypercubeForEntry
 (
  HashTablePtr htp, HashTableEntryPtr ep,
  int partitionRangeMinIndex, int partitionRangeMaxIndex,
  Point entryPoint, int dimension, Point point, Point bucketMidpoint,
  int loopDimensionNum
 );

extern void ComputeVotesInHashTableForEntry
 (
  HashTablePtr htp, HashTableEntryPtr ep,
  int partitionRangeMinIndex, int partitionRangeMaxIndex
 );

extern void ComputeVotesInHashTableEntryListForEntry
 (HashTablePtr htp, HashTableEntryListPtr lp, HashTableEntryPtr ep);

extern Vote Compute_DistanceVote_PredPoint_ExtrPoint
 (
  int dimension, Point predictedPoint, Point extractedPoint
 );

/*
 Compute and return
  log(pdf_predictedPoint(extractedPoint)).
*/
extern float Compute_Log_PDFPtr_PredPoint_ExtrPoint
 (PDFPtr pdfp, Point predictedPoint, Point extractedPoint);

/*
 Compute and return
  backgroundDistrFunc(extractedPoint)
*/
```

```
extern float Compute_BackgroundDistr(Point extractedPoint);

typedef float (* BackgroundDistrFuncPtr)(Point extractedPoint);

/*
 Compute and return
  vote = log
        (
         pdf_predictedPoint(extractedPoint)
          / backgroundDistrFunc(extractedPoint)
        )
*/
extern Vote Compute_Vote_PDFPtr_PredPoint_ExtrPoint_BackgroundDistrFuncPtr
 (
  PDFPtr pdfp, Point predictedPoint, Point extractedPoint,
  BackgroundDistrFuncPtr backgroundDistrFuncPtr
 );

#endif
```

```
#ifndef _matrix_h_
#define _matrix_h_

/*
 matrix.h

 This module implements matrix and vector operations.
*/

#include "utils.h"

/*
 Element of a matrix or vector.
*/

typedef float Element;

typedef Element *ElementPtr;

/*
 Vector
*/

typedef ElementPtr Vector;  /* array dimension of Element */

extern Vector New_Vector(int dimension);

extern void Destroy_Vector(Vector vector);
```

```
#define Get_Vector_ElementI(v, i) \
 (v[(i)])

#define Set_Vector_ElementI(v, i, e) \
 { v[(i)] = (e); }

/*
 b = a
*/
extern void Assign_Vector_Vector(int dimension, Vector a, Vector b);

/*
 c = a + b
*/
extern void Add_Vector_Vector(int dimension, Vector a, Vector b, Vector c);

/*
 c = a - b
*/
extern void Sub_Vector_Vector(int dimension, Vector a, Vector b, Vector c);

/*
 Return 2-norm of a.
*/
extern Element Get_2Norm_Vector(int dimension, Vector a);

/*
 b = e * a
*/
extern void Mult_Vector_Scalar(int dimension, Vector a, Element e, Vector b);

/*
 Return a * b.
*/
extern Element DotProduct_Vector_Vector
 (int dimension, Vector a, Vector b);

/*
 c = a x b
*/
extern void CrossProduct_Vector_Vector_3
 (Vector a, Vector b, Vector c);

/*
 Matrix
*/

typedef ElementPtr Matrix; /* array (numRows*numCols) of Element */

extern Matrix New_Matrix(int numRows, int numCols);
```

```
extern void Destroy_Matrix(Matrix matrix);

#define Get_Matrix_ElementIJ(m, nr, nc, i, j) \
  (m[((i)*nc)+(j)])

#define Set_Matrix_ElementIJ(m, nr, nc, i, j, e) \
  { m[((i)*nc)+(j)] = (e); }

#define Get_Matrix_ElementI(m, nr, nc, i) \
  (m[(i)])

#define Set_Matrix_ElementI(m, nr, nc, i, e) \
  { m[(i)] = (e); }

/* Copy Matrix a to Matrix b. */
extern void Copy_Matrix(Matrix a, Matrix b, int numRows, int numCols);

/* Compute and return the determinant of matrix m. */
/*
extern float det_matrix_3x3(matrix_3x3 m);
*/

/* Compute and return the determinant of matrix m. */
/*
extern float det_matrix_4x4(matrix_4x4 m);
*/

/*------------------------------------------------------------
|
| Routine Name: solve_4x4 - Solve the 4-dim. linear system Ax=z for x.
|
|     Purpose: Solve Ax=z for x, where A is a 4x4 matrix, and
|              z and x are 4x1 vectors.
|
|       Input: z - A 4x1 vector.
|              A - A 4x4 matrix.
|
|      Output: x - A 4x1 vector, the solution.
|
|     Returns: TRUE (1) on success, FALSE (0) otherwise
|
|  Written By: Raju Jawalekar
|        Date: Oct. 23, 1996
| Modifications:
|
-------------------------------------------------------------* /
/*
extern int solve_4x4(vector_4 x, matrix_4x4 A, vector_4 z);
*/
```

```
/*
 Compute w = v*m.
*/
extern void Mult_Vector_Matrix
 (Vector v, int vDim, Matrix m, int mNumRows, int mNumCols, Vector w);

/*
 Compute w = m*v.
*/
extern void Mult_Matrix_Vector
 (Matrix m, int mNumRows, int mNumCols, Vector v, int vDim, Vector w);

/*
 Compute w = v1*v2.
*/
extern void Mult_Vector_Vector(Vector v1, Vector v2, int vDim, ElementPtr wp);

#endif
```

```
#ifndef _utils_h_
#define _utils_h_

/*
 utils.h

 This module implements various utilities - basic data types,
 message printing, etc.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>

typedef char *CharPtr;

#define _MaxNumCharsInString_ (256 - 1)

typedef char String[_MaxNumCharsInString_ + 1];  /* + 1 for '\0' */

typedef int boolean;
#define FALSE 0
#define TRUE 1

#define PI 3.14159265359

extern int _MsgVerbosity_;

/*
```

```
  If not b, then print msg on stdout and exit.
*/
extern void Assert(boolean b, CharPtr msg);

/*
  If msgVerbosityLevel is <= _MsgVerbosity_, then print msg on stdout.
*/
extern void PrintMsg(int msgVerbosityLevel, CharPtr msg);

#endif
```

RUN1 Run Script:

```
ndhash ../targets/targets.txt hum_cargo 0 30 7 16 18 > hum_cargo.0.30.out
ndhash ../targets/targets.txt hum_cargo 0 45 12 20 24 > hum_cargo.0.45.out
ndhash ../targets/targets.txt hum_cargo 0 60 8 18 20 > hum_cargo.0.60.out
tail -32 *hum_cargo*.out > hum_cargo.winners.out

ndhash ../targets/targets.txt m113 0 30 7 10 12 > m113.0.30.out
ndhash ../targets/targets.txt m113 0 45 14 20 25 > m113.0.45.out
ndhash ../targets/targets.txt m113 0 60 8 12 15 > m113.0.60.out
tail -32 *m113*.out > m113.winners.out

ndhash ../targets/targets.txt m35_canvas 0 30 9 18 22 > m35_canvas.0.30.out
ndhash ../targets/targets.txt m35_canvas 0 45 12 22 30 > m35_canvas.0.45.out
ndhash ../targets/targets.txt m35_canvas 0 60 10 20 26 > m35_canvas.0.60.out
tail -32 *m35_canvas*.out > m35_canvas.winners.out

ndhash ../targets/targets.txt m60 0 30 12 16 18 > m60.0.30.out
ndhash ../targets/targets.txt m60 0 45 22 30 36 > m60.0.45.out
ndhash ../targets/targets.txt m60 0 60 14 20 24 > m60.0.60.out
tail -32 *m60*.out > m60.winners.out

ndhash ../targets/targets.txt hum_troop 0 30 17 24 28 > hum_troop.0.30.out
tail -32 *hum_troop*.out > hum_troop.winners.out

ndhash ../targets/targets.txt m35 0 30 12 24 28 > m35.0.30.out
tail -32 *m35.0*.out > m35.winners.out
```

RUN2 Run Script:

```
ndhash ../targets/targets.txt hum_cargo 0 30 7 16 18 10 > hum_cargo.0.30.noise10.out
ndhash ../targets/targets.txt m113 0 30 7 10 12 10 > m113.0.30.noise10.out
ndhash ../targets/targets.txt m35_canvas 0 30 9 18 22 10 > m35_canvas.0.30.noise10.out
ndhash ../targets/targets.txt m60 0 30 12 16 18 10 > m60.0.30.noise10.out
tail -32 *noise*.out > winners.noise.out
```

APPENDIX B.
3D HASH POINT DATA SETS

Synthetic LADAR range images had been previously generated by NYU for another project* from the Ballistic Research Laboratory computer aided design models for six tactical vehicles (targets) using the LARRA/SAIL modeling program. The targets were the M60-A3 tank, M113 armored personnel carrier (APC), M35 truck with a rack and canvas cover, M35 truck without a rack or canvas cover, HMMWV troop version with the conventional sloped rear and HMMWV cargo version which has a square back. These images were generated at a sensor depression angle of zero degrees. Images were created every 15 degrees azimuth and elevation with each pixel corresponding to a ray trace. All output images are in the Khoros viff format.

For these six targets, images at every 30° were selected to use to build up a data base of 72 models consisting of 12 orientations for the 6 targets. Since geometric hashing operates as a recognition/identification process on a subimage corresponding to a region of interest or "image chip", a pseudo background image was created in which to insert these targets. This raw image chip was sized at 100 rows by 150 columns and Gaussian random noise with a mean value of 5 and a variance of 2 was added to every background pixel in this raw image chip. The LARRA/SAIL generated images were subsampled by a factor of three and inserted into the approximate center of this raw image chip. This generated images of the selected vehicles approximated targets at a range of one kilometer.

For the ND Hashing project, NYU extracted 3D hash points from 54 of these LARRA/SAIL-generated images:

- Four targets every 30° in azimuth: HMMWV (cargo), M113 APC, M35 truck (canvas cover), and M60 tank; these represent a 48 model training set.
- The same four targets at a 45° azimuth, for use as test images
- HMMWV (troop) and M35 truck (no canvas), both at 30° azimuth. These two target variants were also used as test images.

The following tables give the (x,y, range) values of each hash point for each of those 54 targets.

* A. Akerman III, R. Patton, W. Delashmit, and R. Hummel, Multisensor Fusion Using FLIR and Ladar Identification, Final Report N-TR-97-131, Army Research Office, Contract DAAH04-93-C-0049, 31 March 1997.

0° Azimuth

HMMWV (Cargo)			M113 APC			M35 Truck (Canvas)			M60 Tank		
x	y	r	x	y	r	x	y	r	x	y	r
3	2	59	5	26	33	5	2	44	2	113	46
87	2	59	43	22	46	101	2	44	21	108	61
88	96	24	55	5	38	103	33	28	15	91	53
76	99	26	66	21	49	96	70	13	27	89	61
75	84	27	94	23	37	98	115	61	47	53	63
15	84	27	105	26	33	79	114	60	66	6	82
14	99	26	105	97	30	79	98	55	75	72	64
2	99	26	91	98	31	71	101	18	103	4	72
			91	83	22	53	102	19	115	77	87
			19	83	22	34	102	21	130	91	53
			19	98	31	27	98	55	127	110	73
			5	98	31	26	114	60	146	113	46
			5	27	32	8	115	61	146	179	36
						10	70	13	118	179	78
						3	34	28	118	163	98
									30	164	98
									30	180	78
									2	180	38

30° Azimuth

HMMWV (Cargo)			M113 APC			M35 Truck (Canvas)			M60 Tank		
x	y	r	x	y	r	x	y	r	x	y	r
3	2	67	9	29	67	5	3	79	7	120	86
105	2	64	84	22	52	165	3	52	69	105	81
118	8	57	109	10	43	165	18	52	77	74	73
119	22	57	100	5	48	175	21	37	99	7	77
137	26	48	131	5	32	177	45	35	106	53	57
139	50	35	106	20	48	189	34	38	128	45	82
150	50	41	176	29	39	197	51	20	139	74	56
166	60	32	189	67	34	198	62	25	147	4	74
163	68	33	187	83	33	207	62	26	173	75	60
160	99	36	173	99	41	214	73	22	190	92	31
144	99	36	26	99	57	214	82	22	255	98	0
136	82	43	11	81	66	204	83	26	195	103	62
107	85	25	6	68	66	198	115	28	258	120	41
96	99	24	11	56	66	186	115	30	262	151	38
77	99	73				177	102	27	240	180	50
65	86	71				162	103	24	23	179	78
38	85	64				144	103	62	6	145	87
30	99	60				135	115	16			
14	99	61				94	115	78			
5	69	66				83	96	57			
						71	114	53			
						31	114	65			
						29	74	67			
						5	63	79			

60° Azimuth

HMMWV (Cargo)

x	y	r
3	2	52
98	2	66
120	8	62
122	24	49
150	26	55
156	50	36
174	50	52
201	60	47
197	68	48
190	99	49
175	99	48
165	89	49
154	99	29
138	99	28
126	82	35
88	82	37
76	99	70
60	99	68
50	88	70
39	99	50
24	99	48
16	70	50
3	69	52

M113 APC

x	y	r
13	27	51
101	22	47
126	18	46
137	11	46
122	3	49
176	4	40
135	24	48
201	28	53
222	62	50
222	79	49
196	98	54
42	98	45
20	89	50
16	79	50
6	77	52
7	68	52
17	56	50

M35 Truck (Canvas)

x	y	r
3	3	58
185	3	60
185	16	60
218	21	48
219	43	48
228	33	51
261	50	33
268	67	41
275	74	41
274	82	39
260	85	42
251	114	43
226	104	23
216	115	24
201	115	25
188	95	27
153	95	40
136	115	66
123	115	64
111	107	45
100	115	47
81	115	71
68	108	53
58	115	54
44	115	52
31	84	53
20	78	62
4	62	58

M60 Tank

x	y	r
3	127	61
31	107	70
87	104	64
78	75	55
93	73	58
107	4	64
115	73	51
123	46	73
128	57	53
149	52	48
172	58	49
202	63	42
175	69	44
239	91	47
365	97	21
237	102	43
230	111	54
301	119	57
316	133	55
308	132	57
310	147	56
272	179	63
41	179	53
11	146	58
11	128	58

90° Azimuth

HMMWV (Cargo)

x	y	r
2	2	36
59	2	36
88	8	36
90	23	36
124	26	36
132	28	37
158	50	42
182	61	43
181	76	36
169	99	36
158	99	36
145	80	36
50	82	38
36	99	36
26	99	36
14	76	36
1	75	44

M113 APC

x	y	r
8	27	40
90	22	47
117	21	47
128	11	50
111	3	50
171	4	50
122	25	48
156	23	58
173	28	35
174	37	35
196	60	39
194	83	34
166	98	61
46	98	34
19	88	34
15	73	33
5	76	34
5	68	34
12	61	34

M35 Truck (Canvas)

x	y	r
1	1	35
156	1	35
158	72	46
163	19	40
204	20	40
207	42	45
264	50	45
263	71	45
269	73	41
269	82	45
254	85	44
240	114	38
224	114	38
206	89	44
128	92	54
118	82	45
99	114	36
75	99	43
52	114	36
31	82	45
4	80	45
2	61	45

M60 Tank

x	y	r
11	107	41
55	103	41
95	107	37
75	99	44
64	74	44
79	74	35
87	4	48
110	73	38
120	57	42
147	52	40
166	57	42
205	63	43
162	71	46
177	69	41
207	81	44
228	93	48
378	98	49
227	102	47
214	114	48
267	119	27
284	132	27
277	138	37
274	151	27
231	179	64
47	179	27
2	127	36
11	119	39

120° Azimuth

HMMWV (Cargo)

x	y	r
4	3	47
96	1	33
121	8	38
123	23	38
158	28	45
158	49	46
174	50	47
202	60	52
198	68	51
192	99	50
176	99	51
165	88	72
155	99	70
140	99	71
127	82	39
88	83	31
77	99	29
62	99	30
51	88	29
41	99	49
25	99	51
15	69	41
4	69	47

M113 APC

x	y	r
13	26	49
101	22	48
125	19	48
137	11	54
122	3	51
174	4	60
134	21	51
201	28	47
202	40	47
222	63	50
222	79	51
192	98	45
44	98	55
18	88	50
16	78	50
6	77	48
6	69	48
17	55	50

M35 Truck (Canvas)

x	y	r
3	3	42
186	2	40
186	14	40
219	21	52
220	43	52
230	33	48
262	50	65
266	66	58
270	72	57
276	75	58
275	82	58
261	85	56
252	115	55
237	115	54
227	104	53
216	115	75
201	115	73
188	94	42
153	96	35
138	114	32
123	114	34
112	108	54
100	115	52
82	115	27
70	107	26
61	114	45
45	114	46
32	84	46
20	77	38
3	61	42

M60 Tank

x	y	r
3	128	41
28	117	32
32	107	29
88	103	19
78	74	43
97	43	41
115	4	32
140	73	26
148	57	34
177	51	34
192	57	35
225	63	46
216	80	42
227	80	47
241	93	57
364	98	78
239	102	49
231	111	43
304	120	42
318	133	45
309	132	43
311	148	44
270	179	36
45	179	48
10	143	42
12	128	42

150° Azimuth

HMMWV (Cargo)

x	y	r
5	1	32
108	2	35
122	9	42
123	23	43
140	26	52
142	49	53
152	50	58
167	60	67
166	69	66
162	99	64
146	99	63
140	83	59
109	84	73
99	99	75
80	99	27
67	85	28
41	85	35
32	99	39
17	99	38
12	70	36
5	69	32

M113 APC

x	y	r
9	27	33
79	22	63
110	12	56
101	5	52
133	5	68
106	19	49
117	25	49
174	30	61
189	63	65
188	83	67
172	99	58
28	99	43
13	89	35
8	78	31
6	67	33
12	54	34

M35 Truck (Canvas)

x	y	r
4	3	21
165	2	48
166	18	48
175	22	63
178	46	65
191	32	61
197	52	79
198	62	69
207	63	74
212	74	76
214	82	78
204	82	73
199	115	70
187	115	68
178	101	73
161	102	75
147	102	78
137	115	82
93	114	20
83	99	49
72	114	43
30	114	32
30	72	18
6	62	21

M60 Tank

x	y	r
7	120	13
19	115	20
37	113	17
69	104	12
61	75	26
77	75	20
83	44	26
98	70	35
114	4	21
125	69	30
160	52	33
176	58	36
186	65	51
179	73	42
194	93	37
204	89	38
255	99	0
199	107	42
228	116	42
260	120	59
266	132	64
263	152	61
239	180	48
27	180	24
6	144	13
5	128	10

180° Azimuth

HMMWV (Cargo)

x	y	r
5	1	22
89	1	22
90	99	29
78	99	29
77	84	72
17	84	72
16	99	29
5	99	29

M113 APC

x	y	r
5	26	21
43	22	31
55	2	52
66	21	49
106	26	21
105	97	31
91	98	32
91	83	23
19	83	23
19	98	33
5	98	33

M35 Truck (Canvas)

x	y	r
4	1	8
101	1	8
102	34	72
96	70	73
98	115	23
81	115	23
79	98	28
67	102	79
53	102	78
39	102	79
27	98	28
27	113	36
7	114	22
10	68	73

M60 Tank

x	y	r
1	113	2
22	107	36
17	90	46
27	91	15
33	76	13
45	2	28
57	70	35
72	74	15
82	4	18
100	52	36
112	58	35
124	78	38
122	88	32
133	92	47
126	105	31
147	113	2
147	180	5
119	180	5
119	163	0
29	163	0
29	180	7
1	178	5

210° Azimuth

HMMWV (Cargo)

x	y	r
65	1	35
168	1	32
168	69	32
161	70	36
156	99	39
140	99	40
132	84	35
105	85	29
93	99	27
74	99	76
63	85	74
33	83	60
27	99	63
11	99	64
7	68	66
5	61	68
20	50	59
31	48	53
34	26	52
50	23	43
51	8	42

M113 APC

x	y	r
21	30	61
80	24	49
84	10	57
64	5	68
96	4	52
94	18	55
113	23	47
187	28	33
185	55	34
189	78	31
171	99	42
23	99	59
10	83	66
7	63	66

M35 Truck (Canvas)

x	y	r
61	3	48
222	2	21
220	63	23
195	73	33
198	114	32
153	114	45
143	99	18
133	114	20
90	115	82
82	103	37
65	103	75
47	102	66
40	115	68
26	114	72
22	82	81
14	78	77
20	63	74
30	61	79
30	51	79
47	47	73
37	33	62
52	22	63
61	18	48

M60 Tank

x	y	r
56	120	59
88	115	42
118	107	42
61	99	0
115	89	38
125	93	56
142	74	38
167	2	27
174	71	42
184	59	40
186	44	17
193	58	39
202	53	42
210	57	36
216	4	23
228	71	37
235	73	25
255	75	26
243	103	28
297	113	20
308	119	14
312	128	10
310	143	13
291	180	23
74	180	50
53	153	61
51	132	65

240° Azimuth

HMMWV (Cargo)

x	y	r
110	1	34
203	2	48
203	69	48
191	69	41
182	99	51
167	99	50
156	88	30
146	99	32
130	99	30
118	82	34
79	83	70
68	99	72
52	99	71
42	88	72
30	99	51
15	99	51
8	67	55
5	60	53
33	50	48
50	50	45
56	26	44
84	23	38
86	7	38

M113 APC

x	y	r
32	28	47
99	23	48
93	9	54
59	4	60
112	3	51
108	19	49
133	22	48
220	27	48
217	55	50
228	69	48
228	77	48
218	77	50
216	88	50
192	98	54
38	98	46
14	82	50
12	63	50
32	39	47

M35 Truck (Canvas)

x	y	r
101	2	40
283	2	42
282	62	41
265	76	33
255	84	46
242	115	48
229	115	45
217	107	26
205	115	27
186	115	53
175	108	54
162	115	35
150	115	33
133	95	58
98	94	46
86	115	73
71	115	75
60	104	53
50	115	54
35	115	55
26	83	65
11	82	58
15	74	65
18	66	58
23	64	56
24	51	63
63	45	57
59	33	48
68	22	52
101	16	40

M60 Tank

x	y	r
36	97	79
161	91	52
175	80	48
227	69	49
198	64	58
228	60	50
244	57	49
251	52	49
274	58	45
277	44	26
284	70	46
294	5	36
308	73	41
322	76	43
313	103	19
378	110	37
379	118	38
399	127	40
389	128	42
391	142	42
360	179	48
128	179	37
92	152	43
92	132	43
83	132	45
97	119	42
153	113	32
169	110	49
162	102	56

270° Azimuth

HMMWV (Cargo)

x	y	r
126	1	37
182	3	37
183	75	45
171	77	37
158	99	37
147	99	37
133	79	51
39	80	37
26	99	37
15	99	37
2	75	44
2	61	44
25	50	43
52	28	61
54	50	37
60	26	37
94	23	37
96	8	37

M113 APC

x	y	r
52	22	41
88	20	48
78	10	50
37	4	50
97	3	50
91	21	47
119	22	47
199	26	40
196	61	34
203	68	34
203	76	34
192	74	40
190	86	34
162	98	61
43	98	34
14	83	34
12	60	39
34	36	35
35	27	35
41	33	34

M35 Truck (Canvas)

x	y	r
116	1	35
271	1	35
268	61	35
268	80	45
240	83	36
222	114	36
197	99	43
174	114	36
154	83	36
144	92	41
71	92	37
61	92	38
49	114	38
33	114	38
20	86	44
2	81	42
4	72	40
11	70	46
7	50	45
66	42	45
68	20	40
109	18	41
115	72	46

M60 Tank

x	y	r
2	97	49
152	90	46
173	80	44
204	70	55
218	70	52
175	64	57
214	58	55
232	52	56
258	57	56
270	69	54
293	4	53
302	74	35
317	75	46
306	98	48
286	105	49
325	103	41
368	107	41
370	120	28
378	127	28
369	141	28
334	178	64
147	178	27
105	150	27
103	138	37
96	132	27
113	119	27
166	112	48
152	102	48

300° Azimuth

HMMWV (Cargo)			M113 APC			M35 Truck (Canvas)			M60 Tank		
x	y	r	x	y	r	x	y	r	x	y	r
107	2	66	32	28	52	102	2	60	36	97	21
201	2	53	84	23	35	284	2	58	162	92	48
201	68	53	100	20	49	280	63	58	175	80	53
189	70	50	93	9	46	267	79	62	204	67	64
181	98	49	58	4	40	256	85	53	177	63	54
165	99	51	112	3	49	243	115	52	208	58	61
155	88	49	109	19	36	229	115	54	226	52	62
145	99	69	133	22	47	218	107	53	254	59	65
129	99	70	221	26	51	206	115	71	263	70	64
117	82	38	217	55	50	187	115	47	286	4	68
78	82	30	227	69	52	177	108	45	305	44	59
66	99	28	228	77	52	163	115	63	310	75	54
51	99	30	218	78	50	151	115	66	324	76	57
40	89	50	216	88	50	135	96	65	312	99	55
30	99	48	189	98	45	99	95	27	370	108	70
14	99	50	41	98	54	87	115	25	373	119	55
6	66	47	14	82	49	71	115	24	389	119	58
4	61	48	12	62	50	62	105	23	399	128	60
31	50	52	32	39	52	51	115	45	389	128	58
48	48	53	32	27	52	35	115	44	391	142	58
49	27	53				26	83	35	357	179	52
56	26	55				12	82	41	132	179	64
82	24	50				17	73	34	92	151	56
85	8	62				23	66	42	93	132	57
						25	50	34	84	132	55
						65	44	39	100	119	57
						59	33	51	155	114	67
						69	21	48	171	111	51
						102	15	60	163	102	43

330° Azimuth

HMMWV (Cargo)

x	y	r
63	2	65
166	2	68
166	69	67
159	70	64
155	99	61
139	99	60
130	85	64
104	86	72
92	99	73
73	99	24
61	85	25
32	82	32
24	99	36
9	99	36
5	66	32
4	60	32
19	50	41
29	50	39
31	27	44
50	24	50
51	8	58

M113 APC

x	y	r
27	29	42
81	25	48
92	18	44
84	7	43
65	5	32
97	5	48
99	18	49
118	22	37
174	27	69
188	28	67
187	58	66
192	68	67
190	78	69
184	89	65
170	99	57
26	99	42
11	85	34
7	66	34
22	36	39

M35 Truck (Canvas)

x	y	r
62	2	52
223	3	79
220	64	78
198	74	67
196	115	66
153	114	54
144	96	57
132	115	77
91	115	17
82	102	22
66	103	24
47	103	34
41	115	30
28	115	28
22	82	19
13	82	22
16	72	24
20	62	26
30	61	20
31	51	20
50	45	35
37	32	38
53	21	36
62	18	52

M60 Tank

x	y	r
61	98	0
115	90	60
126	91	45
139	74	58
132	64	49
150	59	60
158	53	65
165	57	65
185	61	68
192	71	67
203	6	79
219	3	65
234	45	74
241	76	72
257	76	73
247	100	72
280	113	72
297	113	80
297	120	80
309	120	86
310	128	90
309	145	86
289	180	75
78	180	51
55	155	39
52	132	35
58	120	41
91	115	58
119	112	55
120	103	62

45° Azimuth

HMMWV (Cargo)

x	y	r
3	2	60
104	2	65
123	7	60
125	23	60
149	26	52
153	50	37
166	50	46
189	59	40
189	66	39
185	67	40
181	99	42
164	99	42
154	82	49
140	85	28
129	99	26
113	99	26
104	83	28
96	83	68
86	99	71
71	99	71
61	86	69
56	82	43
45	83	52
36	99	55
20	99	55
13	70	57
3	69	60

M113 APC

x	y	r
11	26	59
22	24	64
78	23	67
81	26	37
93	26	33
95	18	51
116	18	45
127	10	44
115	6	48
115	1	48
159	4	35
131	5	43
129	22	48
145	22	46
170	22	46
170	26	46
195	28	46
195	39	46
213	63	42
211	81	41
190	98	47
34	98	52
16	89	57
14	78	58
6	77	61
6	68	61
14	55	58

M35 Truck (Canvas)

x	y	r
3	2	69
181	2	56
181	17	56
203	22	42
204	44	34
215	33	44
234	51	25
237	62	28
247	66	32
252	73	30
253	82	31
240	83	34
232	115	35
218	115	37
206	98	28
200	103	28
193	101	24
181	115	19
167	115	21
154	94	29
139	115	66
124	115	65
114	109	75
105	115	77
73	115	49
63	109	60
54	115	62
39	115	59
28	81	62
23	78	1
26	76	70
3	62	69

M60 Tank

x	y	r
2	128	76
9	121	73
24	120	69
25	112	77
35	112	76
57	105	77
82	105	71
83	97	64
74	95	64
72	76	64
88	74	66
104	75	57
107	6	71
118	58	59
129	57	53
131	45	79
142	55	55
156	4	72
157	59	52
181	64	42
167	74	55
191	75	61
201	81	47
211	81	50
223	94	37
321	98	9
225	103	37
220	113	59
244	113	63
248	119	47
289	120	49
302	134	46
295	133	48
296	152	48
265	180	57
34	180	65
8	143	74
10	129	73

Target Variants: 30° Azimuth

HMMWV (Troop)

x	y	r
16	47	64
43	31	51
84	30	33
84	18	43
73	14	51
91	8	46
95	1	45
104	1	44
105	10	44
110	14	34
94	19	44
93	28	43
104	26	43
125	26	41
146	30	44
151	51	32
162	51	37
178	60	29
176	67	29
172	100	32
156	100	33
150	84	28
118	86	21
109	100	21
90	100	69
78	87	68
51	86	60
41	100	56
26	100	58
22	70	60
15	70	64

M35 Truck (Canvas)

x	y	r
8	12	78
98	11	36
99	3	37
158	2	45
175	6	37
176	31	22
191	16	39
182	31	36
197	36	20
197	47	21
207	47	26
210	53	24
213	58	22
214	67	22
204	68	26
199	100	28
187	100	30
178	86	25
170	82	25
161	88	24
152	82	22
144	88	62
136	100	17
113	98	77
94	100	78
82	81	51
70	100	54
32	100	66
20	79	71
28	58	82
8	51	78

APPENDIX C.

Theoretical Formulation for Hashing of Ladar Imagery

C.1 Model Building with Depth Values and Corner Points

The hash table is constructed that encodes the information about the models in a view-centered fashion. Especially because we are dealing with 3D information, it may be possible to use a different model representation strategy. However, our first object recognition strategy uses separate models for every viewing direction. Accordingly, we begin separate models for each target type, for each discretized viewing direction. The viewpoint direction of the model is a two-parameter collection of locations on the "viewing sphere," although in our initial experiments, we will assume a constant depression angle, and thus the viewpoint direction reduces to a single parameter.

The data that are encoded for each model are of two types: relative depth data and corner discontinuities. That is, for each model, we form two sets of data, using predictions based on the model. One set consists of the depth information at a finely-quantized two dimensional grid of points, resulting in a set $\{(x_i, y_i, z_i)\}$ of depth values. The location of the origin for this collection is unimportant, since the values will only be used in terms of differences. The second set of data consists of locations of corners that are predicted to be visible along depth discontinuities, and can be represented as a collection of two-dimensional locations $\{(x_i,y_i)\}$. The corner data can optionally be attributed with extra information, such as a predicted orientation of the angle bisector of the corner, when projected onto the image plane. In this case, the data takes the form $\{(x_i,y_i,\theta_i)\}$. We reiterate that this information is dependent on the model m, and that a model is a target/orientation pair.

Next, we choose basis sets. A single (x,y,z) location suffices to determine a basis set. Theoretically, we could use all of the depth data as potential basis points, but we instead will limit the size of the hash table and the number of representations of the model by choosing only 3D locations corresponding to corner detections. That is, for every predicted corner location $(\bar{x}_i, \bar{y}_i)$, we find a corresponding (x_{ji}, y_{ji}, z_{ji}) in the depth data that has the same (or nearly the same) (x,y) coordinates, and we consider the index i as a possible basis index for the model m. The actual basis for index i is located at (x_{ji}, y_{ji}, z_{ji}).

We then form hash table entries for the model/basis pair (m,j_i). There are essentially two hash tables, corresponding to the two kinds of data. The depth hash table consists of entries

$$\omega_k(m,i) = (x_k, y_k, z_k) - (x_{ji}, y_{ji}, z_{ji})$$

for all $k \neq j_i$. That is, each position in the model is measured relative to the 3D location of the basis point, and the resulting normalized positions become hash table entries for the particular model with the particular basis.

For the corner data, we construct entries from the predicted observable corners in the Ladar data, normalizing with respect to the (x,y) locations. Thus for every (x_k, y_k, θ_k) encoding a corner location in the model m, we form a hash entry

$$\overline{\omega_k}(m,i) = (\overline{x_k} - x_{ji}, \overline{y_k} - y_{ji}, \theta_k)$$

Thus the corner data entries are relative (x,y) positions with respect to the basis point location, together with the predicted angular bisector direction of the corner.

The entries should additionally be endowed with covariance information; i.e., predictions about the variations of the hash values due to inaccuracies in sensing. This information is needed in order to ensure that the weighted voting geometric hashing scheme properly implements a Bayesian reasoning system, under the assumption that the hash values of the observed scene data provide independent information (a conditional independence assumption). For our preliminary studies, we will use a simplified covariance estimation procedure. Namely, for the hash table entries $\omega_k(m,i)$, we assume a spherical distribution of values centered at the 3D location of the entry, with standard deviation proportional to the Euclidean norm of the hash value entry. For the corner data, the entry $\omega_k(m,i)$ is assumed to have circular variation in the (x,y) components with standard deviation proportional to (but with a larger constant of proportionality) the Euclidean distance from the origin, and the θ component is presumed to be statistically independent and Gaussian distributed with a fixed variance.

C.2 New Voting Schema

Data is obtained on a far coarser sampling rate, and with much greater noise than in the case of the model data. Nonetheless, we are able to extract lines, corners, and have readily available depth values from the observed objects.

We use a corner detector to obtain potential basis points. Currently, we are using the C++ version of the Cox-Boie edge detector, and the line following and coalescing routines. We have ported the Cox-Boie edge detector to KHOROS, displaying the results with Cantana.

In any case, image locations where corners are detected are located. We pick one such point as a candidate basis location (at location, say, (x_0, y_0, z_0)), and we perform a trial. The algorithm must iterate over trials until all interesting locations have been explored. In a trial, we perform hashing of the detected object subimage and weighted voting of the model/basis candidates. Hashing works as follows.

For all pixel locations (x,y,z) near the basis point , (x_0, y_0, z_0) in the scene, we compute a relative $(\xi, \eta, \zeta) = (x,y,z) - (x_0, y_0, z_0)$ value for each such point. The coordinate values correspond to a differential distance from the observed basis point location in the scene. When computing the depth value z_0 for the basis point, we use a local minimum of range values in order to be sure that the range is obtained for the foreground object, and not the background. Each such (ξ, η, ζ) location becomes a hash value that hashes into the three-dimensional range data hash table. We need only concern ourselves with (ξ, η, ζ) values that are sufficiently small that they could plausibly be on the same target as the basis point.

Likewise, nearby extracted corners are used to compute a location (ξ,η, θ) giving a relative position to the basis point and the orientation of the angle bisector. This value hashes into the three-dimensional corner-values hash table.

For each range-based hash, say (ξ, η, ζ), nearby entries are located in the hash table. For each entry of the form $\omega_n(m,i)$ that is located near (ξ, η, ζ) a search is made for the entry $\omega_k(m,i)$ that is closest to (ξ, η, ζ). Since the entries of the form $\omega_q(m,i)$ form a "sheet" representing the surface of the object, they will be located quite densely, and the entry $\omega_k(m,i)$ that is nearest (ξ, η, ζ) will be the nearest point on this surface.

Recall that $\omega_k(m,i)$ is located at $(x_k, y_k, z_k) - (x_{ji}, y_{ji}, z_{ji})$. This entry then receives a vote, which replaces its current vote only if it is greater than its current vote. All votes are initially zero. The vote for entry $\omega_k(m,i)$ is denoted by $z_k(m,i)$, and the vote amount, for the depth data, depends on the distance from the point (ξ, η, ζ) to the sheet, at point $\omega_k(m,i)$. If the (x,y) coordinate locations are far apart, then the observed point is not occurring "in front" of the model, and the vote will be zero. However, ordinarily, if there is one point of the sheet nearby, then the nearest point will be perpendicular to the hash point, which in the nearly orthogonal projection, means that the (x,y) components nearly match. In this case, the distance d is essentially the different in the z components.

The vote should be large if this distance d is small, and will be negative if the distance is large. The Bayesian theory says that the value should be

$$z_k(m,i) = \log\left[\frac{\mathrm{Prob}((\xi,\eta,\zeta)|(m,i),(x_0,y_0,z_0))}{\mathrm{Prob}((\xi,\eta,\zeta))}\right]$$

where the Prob's measure density distribution values at the location of the hash, and the condition in the numerator means that it is known that the model m appears with basis point i at location (x_0, y_0, z_0). To model this vote, we use the formula

$$z_k(m,i) = \log\left(\frac{\frac{1}{\sqrt{2\pi\sigma_1}}e^{-d^2/2\sigma_1^2}}{\frac{1}{\sqrt{2\pi\sigma_2}}e^{-d^2/2\sigma_2}}\right) = c_1 - c_2 d^2$$

where d is the distance between (ξ, η, ζ) and $\omega_k(m,i)$ and σ_1 and σ_2 are constants discussed below.

The value σ_1 is expected depth variation (the standard deviation value, actually) due to sensor noise, measurement noise, and also changes in the vehicle at any given location. The units are in length and so for a high quality sensor, are likely to be on the order of a foot or two. The value of σ_2 is the standard deviation for point to point variations of depth, without any other knowledge. The value of C_1 is $(1/2)\log(\sigma_2/\sigma_1)$, and the coefficient C_2 is simply $(1/2\sigma^2_2)-(1/2\sigma^2_1)$. Presumably, the weighted vote should saturate at some negative amount, and not get too negative, reflecting the fact that a sensor drop-out is possible. Also, this formula could easily be modified to account for the fact that the σ_1 value should be larger for positive values of d,

(representing the possibility of occlusion of the model) than for negative values of d (which would occur when the model has a hole in it).

For hashes of corner detections, a similar formula operates. That is, a hash to location (x,y, θ) is used to locate nearby entries of the form $\overline{\omega}_k(m,i)$. In this case, because corner detections are well separated for any given model/basis combination, there is no need to search for the nearest $\overline{\omega}$ entry with model/basis (m,i). A weighted vote $\overline{z}_k(m,i)$ is recorded for the entry. This time, the "distance" between the hash point and the entry can be measured by a weighted sum of the square distance in the (x,y) plane, and the square difference in the θ variable. The z component plays no role because it has already been accounted in the depth hashes. The weights will depend on the expected variations. Let d^2 represent the weighted sum of square differences. That is,

$$d^2 = a_1\left[(\overline{x_k} - x_{ji} - x)^2 + (\overline{y_k} - y_{ji} - y)^2\right] + a_2(\theta_k - \theta)^2$$

Here, the weights a_1 and a_2 will have to be determined empirically. Then the formula for the weighted vote is similar to before:

$$\overline{z}_i(m, j_i) = \overline{c_1} - \overline{c_2}d^2$$

Again, the value should be clipped if it becomes too negative. Also, only corners near the basis point need be considered. Here, the $\overline{C}_1$ and $\overline{C}_2$ values depend on two standard deviation values, σ_1 and σ_2, just as above, where the first represents expected distances of the corners from nearby corner entries given knowledge of the placement of the model, and the σ_2 entry corresponds to a priori distance deviations.

Finally, votes are combined. The total weighted vote for any given model/basis is a sum of the weighted votes for all entries bases on the model/basis:

$$W(m, j_i) = \sum_i z_i(m, j_i) + \sum_i \overline{z}_i(m, i)$$

This sum is performed over all model/basis sets, and model/bases that receive a large weighted vote are candidate detections.

The result is that a model that is likely to be present will receive a large corresponding vote for some (m,i) pair, providing the chosen basis location , (x_0, y_0, z_0) lies near a corner of a model point. We thus see that it is extremely important to be able to extract from the detected subimage basis points (in our case, corner points) that correspond to corner points pre-stored as basis points in the models.